MIKELANGELO

D5.3

Final report on the Integration of sKVM and OSv with Cloud and HPC

Workpackage	5	Infrastructure Integration	
Author(s)	John Kennedy, Marcin Spoczynski		INTEL
	Peter Chronz, Maik Srba		GWDG
	Nico Struckmann, Yosandra Sandoval		HLRS
	Miha Pleško, Justin Činkelj, Gregor Berginc		XLAB
Reviewer	Jochen Buchholz		HLRS
Reviewer	Gregor Berginc		XLAB
Dissemination Level	PU		

Date	Author	Comments	Version	Status
2017-10-18	John Kennedy	Initial outline	V0.0	Draft
2017-12-12	All	Initial contributions complete	V0.1	Draft
2017-12-13	John Kennedy	Ready for review	V0.2	Review
2017-12-22	All	Post-review modifications complete	V1.0	Final

Executive Summary

The MIKELANGELO project has sought to improve the I/O performance and security of Cloud and HPC deployments running on the OSv and sKVM software stack. These improvements have been realised by the development of new software components in parallel with the enhancement of existing software projects. In both Cloud and HPC deployments, a substantial set of software needed to be installed, configured and integrated. This document summarises the Infrastructure Integration activities that have taken place during the third year of the MIKELANGELO project.

Some integration efforts have been independent of deployment environment. The full-stack instrumentation and monitoring system built on INTEL's Snap open-source telemetry project has been expanded and refined during year three of the project as the testbeds have matured and enhanced use cases have been deployed by MIKELANGELO. The Lightweight Execution Environment Toolbox now enables OSv-based unikernel workloads to be hosted by Kubernetes.

The Scotty continuous experimentation framework has been streamlined and now supports the complete experiment life-cycle up to and including automated statistical data analysis across experimentation runs. It is complemented by Actuator.py that provides an extensible framework for configuring software components, and the MIKELANGELO Cloud Manager which provides a layer of abstraction above infrastructure management and telemetry systems to support the research and development of new scheduling and orchestration algorithms.

To evaluate MIKELANGELO components in realistic Cloud and HPC environments, the OpenStack testbed at GWDG and the Torque-based testbed at USTUTT have both been extensively updated and enriched. They now provide production-like environments to host the MIKELANGELO use cases with enhanced extensions to OpenStack and Torque developed to facilitate automation and experimentation. Many MIKELANGELO components including IOcm, vRDMA, Snap, SCAM, OSv, LEET, Scotty, Actuator, MCM and vRDMA have been successfully integrated into these environments.

The integrated infrastructure is built extensively on open-source software. New components and enhancements to existing software developed by MIKELANGELO are being open-sourced where possible. Relevant contributions have been submitted to complementary third-party open-source projects.

Acknowledgement

The work described in this document has been conducted within the Research & Innovation action MIKELANGELO (project no. 645402), started in January 2015, and co-funded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme (H2020-ICT-07-2014: Advanced Cloud Infrastructures and Services)

MIKELANGELO

Table of contents

Table of Figures

Table of Tables

1 Introduction

The MIKELANGELO project has sought to improve the I/O performance and security of Cloud and HPC deployments running on the OSv and sKVM software stack. These improvements have been realised by the development of new software components, as well as the enhancement of existing software projects. In some cases enhancements in disparate projects needed to be synchronised. In both Cloud and HPC deployments, a substantial set of software needed to be installed, configured and integrated.

This document describes the steps taken in the third year of MIKELANGELO to deliver fully integrated infrastructure. These activities are managed within Work Package 5 Infrastructure Integration which has three tasks: T5.1 Management in Cloud Environments, T5.2 High Performance Computing, and T5.3 Instrumentation and Monitoring. This work package and all its three tasks were scheduled to complete in Month 36 of the project - December 2017.

An overview of infrastructure integration within the MIKELANGELO project is presented in Chapter 2. It introduces the various software components that need to be integrated on both the Cloud and HPC testbeds, and the various enabling components that needed to be developed as part of this work.

Chapter 3 describes the various enabling components that needed to be developed to realise a completely integrated infrastructure. This includes the Instrumentation and Monitoring framework, the Lightweight Execution Environment Toolbox, the Scotty Continuous Experimentation framework, the Actuator.py framework for resource manipulation and the MIKELANGELO Cloud Manager designed to manipulate arbitrary cloud infrastructure.

Chapter 4 focuses on Cloud Integration, and details the advances made in the production-like Cloud testbed hosted by GWDG during the third year of the project. This testbed is constructed using OpenStack and is designed to support a wide range of Cloud and Big Data workloads. The integration of MIKELANGELO components is explained.

Chapter 5 focuses on HPC Integration, and describes the year 3 enhancements made to the HPC testbed hosted by HLRS. Built on the open source Torque scheduling system, this testbed has been built to demonstrate and evaluate how HPC workloads can be deployed on MIKELANGELO-optimised Cloud infrastructure. Integration with the various MIKELANGELO components including Scotty is detailed.

Chapter 6 summarises some key observations on the various components progressed by the infrastructure integration efforts. Chapter 7 provides some concluding remarks. References are provided in Chapter 8. The Appendix lists various configuration options for vTorque.

2 Integration Overview

2.1 MIKELANGELO Architecture

There are numerous independent software platforms, frameworks and components that needed to be successfully integrated to realise MIKELANGELO's vision of substantially improved performance and security of virtualised environments for both Cloud and HPC. These various systems are presented and described in Deliverable D2.21, The Final MIKELANGELO Architecture[1]. The following figure illustrates all key components that make up the MIKELANGELO architecture at the end of the project.

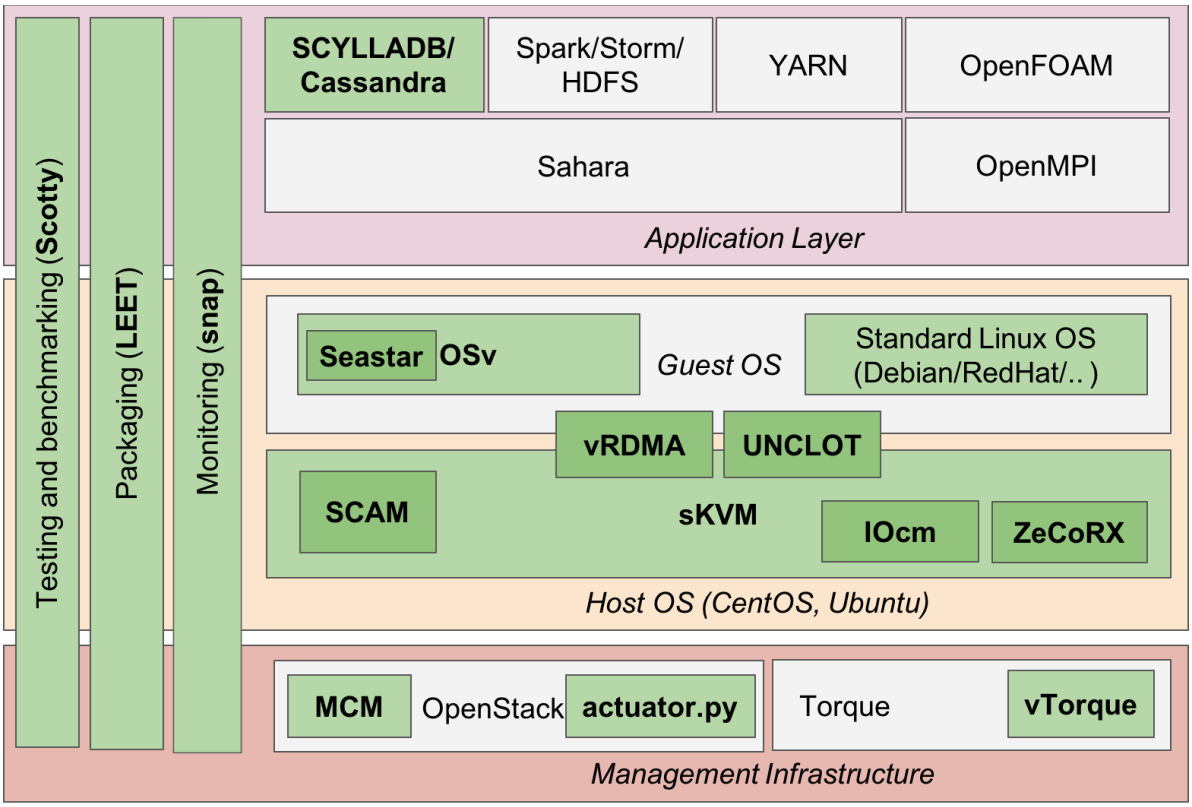


Figure 1: High-level Cloud and HPC-Cloud architecture at the end of the project.

A core component of MIKELANGELO is the Super KVM (sKVM) hypervisor - receiving dedicated security - side-channel attack mitigation (SCAM) - and performance enhancements including the IO Core Manager (IOcm), and Zero Copy Receive (ZeCoRx). Another core component is the OSv Guest Operating System, receiving numerous performance optimisations. Across these two components, various optimisation techniques such as Unikernel Cross Level Communication Optimisation (UNCLOT) and Virtio Remote Direct Memory Access (vRDMA)-based performance enhancements have been delivered.

To demonstrate a complete cloud stack, these core components have been integrated with

the OpenStack cloud management platform. The MIKELANGELO Cloud Manager component has been developed to support the live provisioning of workloads on the cloud platform of choice - initially OpenStack is supported. The manipulation of the various components in the complete stack is managed using Actuator and its plugins.

To demonstrate the advantages of applying MIKELANGELO components to HPC workloads, the MIKELANGELO core components have been integrated with the vTorque resource manager. This extends the Torque HPC bare-metal resource manager with support for virtual machines.

Complementing these internal technical advancements, the practicalities of managing complex virtualized stacks across realistic multi-node use-cases across both Cloud and HPC infrastructures have also being addressed. The Lightweight Execution Environment Toolbox (LEET) helps automate the packaging of cloud and HPC workloads, as well as deploying them on various infrastructures. In order to understand how each and every component of the integrated stacks are performing, Snap provides an extensible full-stack monitoring system. Scotty delivers a Continuous Experimentation framework inspired by Continuous Integration techniques, and helps automate the complete lifecycle of evaluating a change to a component on the software stack.

This deliverable explains the year three advancements in the integration-enabling components of MIKELANGELO including the Instrumentation and Monitoring stack and the Continuous Experimentation Framework. It also describes the progress the integration of the various components in both the Cloud and HPC stacks. All of these advancements were progressed in response to the Infrastructure Integration requirements.

2.2 Infrastructure Integration Requirements

The various deployment scenarios and requirements for MIKELANGELO were tracked in MIKELANGELO's Work Package 2, Use Case and Architecture Analysis. By the end of the project over 55 specific scenarios regarding Infrastructure Integration have been documented as of the time of writing, resulting in 53 consolidated requirements categorised as infrastructure or monitoring related. This list grew as new needs were identified, as new use cases were deployed on the evolving Cloud and HPC testbeds.

Requirements reflected the need to support deployment and configuration of the various MIKELANGELO and use case software components. Configuration and instantiation of appropriate instrumentation and monitoring also featured heavily, with the need for relevant data to be gathered in appropriate back-end systems for offline analysis. As new software components were introduced via the use-cases, the need to collect data from these new sources needed to be addressed and also generated numerous requirements.

The list of requirements was regularly reviewed, and drove the prioritisation of activities within Work Package 5, Infrastructure Integration. This work has been organised into generic integration enablers, Cloud-specific integration, and HPC-specific integration as detailed in the following chapters. The requirements for various components are introduced in the relevant sections.

3 Integration Enablers

MIKELANGELO has constructed both a cloud and a HPC environment to help demonstrate and evaluate the MIKELANGELO stack in different, realistic scenarios. A number of core capabilities were required by both deployments, and so a set of components were developed to be standalone, and readily integratable into both testbeds. These included Instrumentation and Monitoring (via Snap), Lightweight Execution Environment Toolbox, a Continuous Experimentation Framework (via Scotty), Actuator.py and MCM: a live resource manager. These various integration enablers are now introduced.

3.1 Instrumentation and Monitoring

3.1.1 Requirements

The fundamental requirements of the instrumentation and monitoring system required by MIKELANGELO were established in the first year of the project and documented in Deliverable D5.7, First report on the Instrumentation and Monitoring of the complete MIKELANGELO software stack[2]. For reference, the initial requirements included:

- the ability to capture hardware metrics
- the ability to capture hypervisor metrics
- the ability to capture guest OS metrics
- the ability to capture hosted application and services metrics
- support for a rich monitoring GUI
- minimal performance overhead on the systems being monitored
- the ability to scale to deployments with thousands of nodes
- the ability to readily manage deployments of thousands of nodes
- the ability to integrate with existing and future tools and technologies to capture data, process data or store data
- the ability to readily adjust the precise metrics being gathered and at what frequency
- a high degree of trustworthiness in the data collected
- a suitable open-source licensing model to enable community adoption and engagement

Following development of the initial Snap framework and its public open-source release in December 2015, the second year of MIKELANGELO additional requirements focusing on supporting the collection of metrics of particular interest to the various testbeds and use cases deployed by the partners of the project.

Additionally, some new core functionality requirements were also identified and addressed. The new requirements tackled by MIKELANGELO during the second year of the project included:

- the ability to monitor OpenFOAM jobs
- the ability to monitor vRDMA processes
- the ability to monitor MongoDB databases
- the ability to monitor the Linux scheduler
- the ability to monitor IOcm status
- the ability to monitor Spark deployments
- the ability to monitor CloudSuite Data Caching
- the ability to tag monitored data with metadata (e.g. an identifier for the experimental run underway, or some details of system configuration)
- the ability to dynamically reduce the resolution of monitoring data unless an anomaly is detected
- the ability to capture Utilisation, Saturation and Error information as powerful high level abstractions for key subsystems such as CPU, memory and I/O
- the ability to automatically monitor additional VMs and virtual devices as they are created in a libvirt deployment
- reducing the overhead of collecting metrics from libvirt
- maintaining compatibility of all MIKELANGELO-developed Snap plugins with the latest releases of the Snap telemetry framework as it evolved towards version 1.0, released November 2016.

These requirements were addressed in the second year of the project, with details of progress documented in Deliverable D5.2, Intermediate Report on the Integration of sKCM and OSv with Cloud and HPC[3].

By the third year of the project, the priority instrumentation and monitoring requirements of MIKELANGELO had been met. However, the need for some incremental enhancements did arise, and opportunities were also found to refine the compatibility, performance and usability of the overall system. The key additional requirements that emerged included:

- the ability to monitor KVM deployments
- the ability to monitor OpenVSwitch deployments
- simplifying the deployment of Snap via a dedicated installation and configuration application
- the ability to display diagnostic information about the status of the plugin
- maintaining compatibility of all MIKELANGELO-developed Snap plugins with the latest releases of the Snap telemetry framework as it evolved towards version 2.0, released September 2017.

Opportunities also arose to incrementally enhance the existing plugins and core components. Bugs, sometimes in third party dependencies, were identified and removed. The opportunity to refactor the Snap API using the Swagger 2.0 OpenAPI was seized, and the ability to collect and process events from streaming sources was added.

Whilst INTEL maintained responsibility for the overall instrumentation and monitoring framework in MIKELANGELO, it is worth noting that other partners in MIKELANGELO, specifically IBM and GWDG, have been able to successfully develop and release Snap plugins themselves to expose metrics of particular interest to them. IBM developed the IOcm collector and GWDG developed collectors for both Spark and CloudSuite Data Caching.

3.1.2 Architecture

As introduced in D5.7, First report on the Instrumentation and Monitoring of the complete MIKELANGELO software stack, Snap is an open-source telemetry framework specifically designed to help simplify the gathering and processing of rich metrics within a data center. With deeper instrumentation and analysis of such infrastructure and hosted applications, more subtle measurements of performance can be gathered, and more efficiencies can be realised. The architecture of Snap has evolved somewhat since then. An understanding of some basic aspects of Snap is useful when considering developments in Year 3 and so the latest architecture of Snap is presented here for completeness.

As an extensible and open telemetry gathering system, Snap aims to provide a convenient and highly scalable framework from which arbitrary metric-collecting systems, analytics frameworks, and data stores can be leveraged. Thus, full stack monitoring is achievable, allowing data from hardware, out-of-band sources such as Node Manager, DCIM and IPMI to be analysed together with data from the host operating systems, hypervisors, guest operating systems, middleware and hosted applications and services. These metrics can be transformed or filtered or be processed using any external tool locally, before being published to any destinations, be they local or remote.

Snap is organised into several core components as illustrated in the following figure.

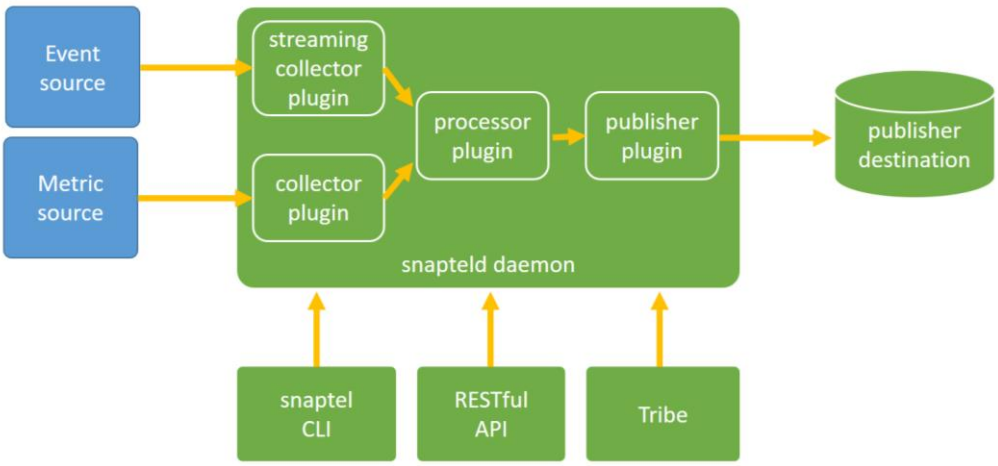


Figure 2: Core components of Snap.

snapteld (previously snapd) is a daemon that runs on nodes that collect, process and publish telemetry information. snapteld may collect the information from the local node (localhost), or the information may be captured from remote nodes, over the network. The latter allows out-of-band metrics to be captured from systems that are in the powered down state. The collection, processing and publishing of telemetry information is done through a flexible and dynamic plugin architecture that is described later in this chapter. This daemon also schedules the tasks that define what data is collected, how it is processed and published. This daemon has a RESTful API.

snaptel (previously snapctl) is a command line interface that allows Snap to be managed. It allows Snap metrics, plugins and specific monitoring instructions known as tasks to be queried and manipulated as required. All three of these elements can also be manipulated via a RESTful API.

Snap has a flexible plugin architecture. To facilitate administration plugins can be versioned and signed. Four fundamental types of plugins are now supported:

Collector plugins allow data to be fed into Snap from a particular source. The metrics exposed by a collector plugin are added to a dynamically generated catalogue of all available metrics. A collector plugin may be engineered to capture data from any source including hardware, operating system, hypervisor, or application source. The data may come from out-of-band systems such as baseboard management controllers - essentially small dedicated always-on CPUs that allow computers to be managed remotely via a dedicated network channel - even if the main system is powered off. The data may come from data-center utility systems – or indeed sources outside the data-center.

Streaming-Collector plugins are a new type of collector plugin that use grpc streams to feed new telemetry data into Snap immediately, within a predetermined amount of time, or when a particular event-count has been reached. This functionality is designed to allow event-based Snap workflows to be constructed, important for runtime orchestration and optimisation applications. Whereas standard collector plugins are configured to transmit data at a particular rate, this new plugin type allows data to be forwarded and processed in the normal way as soon as it becomes available, or within a guaranteed amount of time, or when a certain event-count is reached.

Processor plugins allow Snap telemetry data to be queried and manipulated before being transferred. A processor plugin can be used to encrypt the data, or perhaps convert the data from one format to another. Data can be cached or filtered or transformed – e.g. into rolling averages.

Publisher plugins are used to direct telemetry data to a back-end system. Data could be published to a database, to a message bus, or directly to an analytics platform. Destinations may be open source (e.g. PostgreSQL, InfluxDB) or proprietary (SAP HANA).

Available plugins are loaded into the Snap framework dynamically, and exposed functionality is available without needing to restart any service or node. Once plugins are loaded into the local Snap daemon, specific workflows called tasks can be defined to detail what data is gathered where, and how it is processed and shared. The currently available plugins are listed in the plugin catalogue[4]. At the time of writing (December 2017) there are 100 open-source Snap plugins available from the main Snap repository on GitHub[5]. These include 64 Snap collector plugins, 2 Snap streaming-collector plugins, 10 Snap processor plugins, and 24 Snap publisher plugins. Twelve of these 100 plugins have been contributed and open-sourced from the MIKELANGELO project. Already other members of the open-source community have started to build on these and open-source their contributions. Plugins that are not yet relevant to the broad Snap user base are typically open sourced elsewhere. MIKELANGELO has open-sourced two collector plugins which fall into this category via the MIKELANGELO GitHub repository. These collect data from IOcm and vRDMA - systems which for now are only relevant to Snap users that have installed the MIKELANGELO IOcm and vRDMA solutions.

Table 1 lists the plugins that have been developed and open-sourced by the MIKELANGELO project to date.

Table 1: Snap plugins contributed by MIKELANGELO as of December 2017, Published (X) or Enhanced (x)

Plugin Type	Plugin Name	Plugin Description	2015	2016	2017	GitHub
Collector	Libvirt	Captures data from libvirt	X	x	x	Snap
	OSv	Collects from OSv	X	x	x	Snap
	MongoDB	Captures data from MongoDB		X	x	Snap
	SCSI	Captures data from SCSI devices		X	x	Snap
	OpenFOAM	Captures data from OpenFOAM		X	x	Snap
	yarn	Captures data from yarn scheduler		X	x	Snap
	schedstat	Captures data from Linux scheduler		X	x	Snap
	USE	Captures Utilisation, Saturation and Errors information from various system components		X	x	Snap
	OVS	Captures data from Open VSwitch deployments			X	Snap
	vRDMA	Captures data from vRDMA process		X	x	MIKELANGELO
	IOcm	Captures IOcm data		X	x	MIKELANGELO
	KVM	Captures data from KVM			X	Snap
Processor	Tag	Allows metrics to be tagged		X	x	Snap

	AnomalyDetection	Dynamically scales resolution of telemetry to reduce system overhead		X	x	Snap
Publisher	PostgreSQL	Writes to PostgreSQL database	X	x	x	Snap

Additionally, collector plugins for Spark and CloudSuite Data Caching were developed and deployed locally in the GWDG Cloud Testbed.

All of the plugins produced by MIKELANGELO had to be refactored somewhat during 2017 as the core Snap framework matured into release 2.0 and interfaces stabilised. Those Snap components that were enhanced significantly or first released in 2017 are now introduced.

3.1.3 Collector Plugins

3.1.3.1 KVM Collector Plugin

MIKELANGELO uses the Snap KVM collector plugin to collect metrics from the KVM Hypervisor[6]. This open-source hypervisor is very popular in cloud deployments, is the hypervisor which MIKELANGELO has been enhancing to create sKVM, and is the default hypervisor used by both OpenStack[7] and vTorque[8] implementations as used in MIKELANGELO. This plugin has been developed in order to expose the KVM-specific metrics. The Snap libvirt Collector Plugin[9], which instruments the libvirt abstraction layer above the hypervisor, is not able to extract this level of detail.

The KVM collector plugin can collect all system debug statistics exposed by KVM. These statistics have been mapped to the Snap namespace as follows:

```
/intel/kvm/insn_emulation
/intel/kvm/insn_emulation_fail
/intel/kvm/invlpg
/intel/kvm/io_exits
/intel/kvm/irq_exits
/intel/kvm/irq_injections
/intel/kvm/irq_window
/intel/kvm/largepages
/intel/kvm/mmio_exits
/intel/kvm/mmu_cache_miss
/intel/kvm/mmu_flooded
```

```
/intel/kvm/mmu_pde_zapped
/intel/kvm/mmu_pte_updated
/intel/kvm/mmu_pte_write
/intel/kvm/mmu_recycled
/intel/kvm/mmu_shadow_zapped
/intel/kvm/mmu_unsync
/intel/kvm/nmi_injections
/intel/kvm/nmi_window
/intel/kvm/pf_fixed
/intel/kvm/pf_quest
/intel/kvm/remote_tlb_flush
/intel/kvm/request_irq
/intel/kvm/signal_exits
/intel/kvm/tlb_flush
/intel/kvm/efer_reload
/intel/kvm/exits
/intel/kvm/fpu_reload
/intel/kvm/halt_attempted_poll
/intel/kvm/halt_exits
/intel/kvm/halt_successful_poll
/intel/kvm/halt_wakeup
/intel/kvm/host_state_reload
/intel/kvm/hypercalls
```

To use, the user needs only to subscribe to the list of metrics of interest, as illustrated in the following example task manifest file:

```
version: 1
  schedule:
    type: "simple"
    interval: "1s"
  max-failures: 10
  workflow:
    collect:
      metrics:
        /intel/kvm/insn_emulation: {}
        /intel/kvm/insn_emulation_fail: {}
        /intel/kvm/invlpq: {}
        /intel/kvm/io_exits: {}
        /intel/kvm/irq_exits: {}
      publish:
        - plugin_name: "file"
          config:
            file: "/tmp/kvm_metrics"
```

The location of the plugin within the MIKELANGELO stack is highlighted in yellow in the following figure. Complete details of the plugin can be seen in GitHub at https://github.com/intelsdi-x/snap-plugin-collector-kvm.

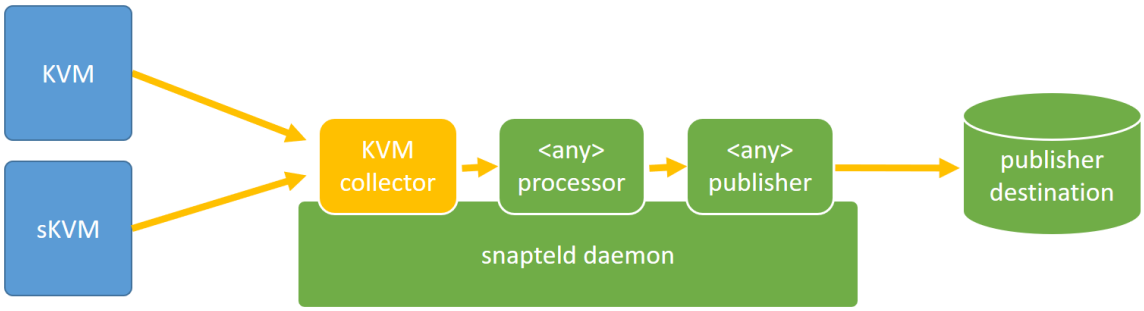


Figure 3: KVM collector plugin within the MIKELANGELO stack

3.1.3.2 Open vSwitch Collector Plugin

MIKELANGELO uses the Snap Open vSwitch collector plugin to collect metrics from the Open vSwitch database. This open-source software-defined switch is one of the most popular production quality, multilayer virtual switch implementation and is the default network solutions used by both OpenStack and vTorque. This plugin has been developed in order to expose the Open vSwitch metrics, like number of flows, packets and bytes transmission per flow and bridge. This knowledge is necessary to measure performance and understand resource utilization of the OVS process, which can cause saturation of machine due to high value of L2 and L3 cache misses spend on packet processing.

The Open vSwitch collector plugin can collect all Open vSwitch standard network interface metrics. These statistics have been mapped to the Snap namespace as follows:

```
/intel/ovs/rx_packets
/intel/ovs/rx_bytes
/intel/ovs/tx_packets
/intel/ovs/tx_bytes
/intel/ovs/rx_dropped
/intel/ovs/rx_frame_err
/intel/ovs/rx_over_err
/intel/ovs/rx_crc_err
/intel/ovs/rx_errors
/intel/ovs/tx_dropped
/intel/ovs/collisions
/intel/ovs/tx_errors
```

```
/intel/ovs/ingress_policing_rate
/intel/ovs/ingress_policing_burst
/intel/ovs/cfm_health
```

Note that the three last metrics indicate health or limits applied to the overall Open vSwitch instance itself. These metrics are:

- /intel/ovs/ingress_policing_rate: returns the maximum rate for data received on this interface, in kbps.
- /intel/ovs/ingress_policing_burst: returns the maximum burst size for data received on this interface, in kb.
- /intel/ovs/cfm_health: Indicates the health of the interface as a percentage of Continuity Check Messages frames received.

To use the plugin, the user needs only to subscribe to the list of metrics of interest, as illustrated in the following example task manifest file:

```
version: 1
  schedule:
    type: "simple"
    interval: "1s"
  max-failures: 10
  workflow:
    collect:
      metrics:
        /intel/ovs/rx_packets: {}
        /intel/ovs/rx_frame_err: {}
        /intel/ovs/rx_crc_err: {}
        /intel/ovs/collisions: {}
        /intel/ovs/tx_errors: {}
      publish:
        - plugin_name: "file"
          config:
            file: "/tmp/ovs_metrics"
```

The location of the plugin within the MIKELANGELO stack is highlighted in yellow in the following figure. Complete details of the plugin can be seen in GitHub at https://github.com/intelsdi-x/snap-plugin-collector-ovs.

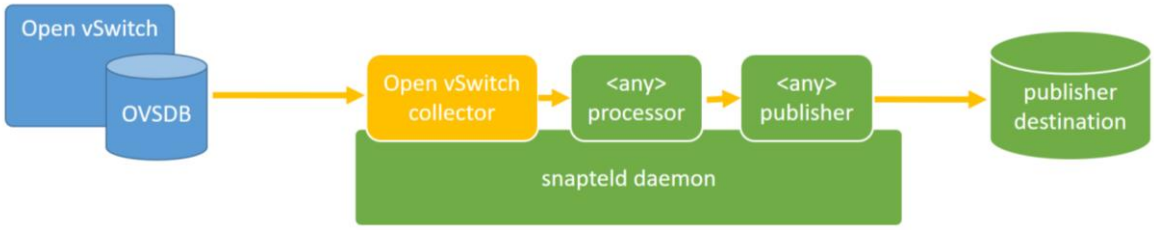


Figure 4: Open vSwitch collector plugin within the MIKELANGELO stack

3.1.4 Snap Deploy

Snap-deploy is software that automates provisioning and configuration management of the Snap framework and plugins.

Configuration of telemetry software often requires a lot of manual effort and complicated coordination. Snap-deploy addresses this issue by providing a convenient and consistent way of configuring, re-configuring and managing Snap deployments. Snap-deploy is a binary application. It is open-sourced via GitHub[10].

The snap-deploy implementation includes the following functionality:

- Deploy - configure and deploy the Snap framework with appropriate plugins
- Redeploy - reconfigure and re-deploy the Snap framework with appropriate plugins
- Download - download Snap framework binaries and appropriate plugins
- Kill - kill the Snap service process along with plugins
- Start - start the Snap process and configured tasks
- Generate task - create a task manifest and export as a json file
- Help - print user help

To use, the user first provides configuration values. Users can configure options including the:

- Time-series database hostname along with database port
- Time-series database username - default: "snap"
- Time-series database password - default: "snap"
- Metric collection interval - default: 1s
- Custom tags that can be added to the measurement
- List of the metrics which should be collected
- Directory where Snap binaries and plugins will be installed
- List of the plugins that will be downloaded and installed

In typical usage, the user needs only to provide configuration options for the back-end time series database, the required metrics, and the required plugins list as illustrated in the following example configuration:

```
$ snap-deploy deploy --dbhost influxdb.hostname.eu --dbdatabase snap --
dbuser snap  --dbpassword snap --interval 1s --metrics "/intel/*" --
directory /opt/snap --plugins collector-psutil, publisher-influxdb
```

Alternatively, configuration options can also be provided as environment variables, as follows:

```
export DB_HOST=influxdb.hostname.eu
export DB_NAME=snap
export DB_USER=snap
export DB_PASS=snap
export INTERVAL=1s
export PLUGINS="collector-psutil,publisher-influxdb"
export METRICS="/intel/*"
export DIRECTORY="/opt/snap"
export TAGS="rack1:rack,datacenter:dublin,experiment:1"
snap-deploy deploy
```

Complete details of the snap-deploy application can be seen in GitHub at https://github.com/intelsdi-x/snap-deploy.

3.1.5 General enhancements

Numerous general enhancements have also been contributed to the Snap core and suite of plugins as driven by the needs of MIKELANGELO and its use-cases.

During deployment on the various testbeds it became obvious that some sort of diagnosis functionality would be useful to assist with validation of the integrity of the Snap installation and configuration.

To address this need, enhancements were made to our snap-plugin-libs libraries to deliver a generic diagnostic functionality.

The new plugin diagnostics now provides a simple way to verify that a plugin is capable of running without requiring the plugin to be loaded and a task started. This feature works for collector plugins written using one of our new snap-plugin-libs.

Plugin diagnostics delivers the following information:

- runtime details (plugin name, version, etc.)
- configuration details
- a list of metrics exposed by the plugin
- metrics with their values that can be collected right now
- the time required to retrieve this information

To view the diagnostic information, the plugin just needs to be executed. Some plugins require a configuration to be supplied. Figure 5 illustrates the output of simply invoking the Snap psutil collector plugin.

```
Runtime Details:
    PluginName: psutil, Version: 14
    RPC Type: gRPC, RPC Version: 1
    Operating system: linux
    Architecture: amd64
    Go version: go1.8.3
printRuntimeDetails took 27.779µs

Config Policy:
NAMESPACE               KEY             TYPE            REQUIRED        DEFAULT         MINIMUM         MAXIMUM
intel.psutil.disk       mount_points    string          false
printConfigPolicy took 181.711µs

Metric catalog will be updated to include:
    Namespace: /intel/psutil/load/load1
    Namespace: /intel/psutil/load/load5
    Namespace: /intel/psutil/load/load15
    Namespace: /intel/psutil/cpu/*/nice
    Namespace: /intel/psutil/cpu/cpu-total/nice
    Namespace: /intel/psutil/cpu/*/iowait
    Namespace: /intel/psutil/cpu/cpu-total/iowait
    Namespace: /intel/psutil/cpu/*/steal
    Namespace: /intel/psutil/cpu/cpu-total/steal
    Namespace: /intel/psutil/cpu/*/guest
    Namespace: /intel/psutil/cpu/cpu-total/guest
    Namespace: /intel/psutil/cpu/*/guest_nice
    Namespace: /intel/psutil/cpu/cpu-total/guest_nice
    Namespace: /intel/psutil/cpu/*/stolen
    Namespace: /intel/psutil/cpu/cpu-total/stolen
    Namespace: /intel/psutil/cpu/*/system
    Namespace: /intel/psutil/cpu/cpu-total/system
    Namespace: /intel/psutil/cpu/*/idle
    Namespace: /intel/psutil/cpu/cpu-total/idle
    Namespace: /intel/psutil/cpu/*/softirq
    Namespace: /intel/psutil/cpu/cpu-total/softirq
    Namespace: /intel/psutil/cpu/*/user
```

Figure 5: Output from the Snap psutil collector plugin diagnostic interface

It was also realised that for some occasions and sparse events it would be useful to process and publish these occurrences immediately, rather than wait a predetermined poll-time. Thus a new plugin type - Streaming Collector - was conceived. Streaming collector plugins can now use grpc streams to allow the plugin to send data immediately or within a certain period of time, instead of on an interval governed by Snap. Streaming by Snap now enables:

- Improved performance by enabling event based data flows
- Runtime configuration controlling event throughput
- Buffer configurations which dispatch events after a given duration and/or when a given event count has been reached

Finally, the Snap APIs have been redeveloped to support a V2 REST API built on top of the Swagger 2.0 OpenAPI framework. The new Snap API was redesigned from scratch to provide a simpler way of interacting with common software tools including:

- go-swagger - Swagger is a simple yet powerful representation of application API, and was used to build new Snap API. It allows API clients to be rapidly developed in a wide range of programming languages.
- swagger-ui - allows the visualisation of and interaction with the API's resources without having any of the implementation logic in place.
- libcurl - a free, multiplatform and easy-to-use client-side URL transfer library.
- Rest clients - api testing tools .
- Snaptel CLI (snaptel) - the Snap command line client.

All of these enhancements were supported by MIKELANGELO contributions to facilitate the deployment, performance and usability of the framework.

3.1.6 Integration and Testing

In year three of the project additional effort was put into streamlining the Snap development and integration process to maximise robustness of the overall codebase. Travis CI was selected to manage this continuous integration process.

Travis CI is a hosted, distributed continuous integration service used to build and test software projects hosted at GitHub. It supports multiple platforms and languages. All MIKELANGELO contributions to public Snap repositories use Travis to unit test, report code coverage and finally, build and release binaries on AWS s3 storage.

Figure 6: Standard workflow for Snap plugin development cycle with Travis CI

Travis CI provides a default build environment and a default set of steps for each programming language. To integrate the Github review system with Travis, the user needs to provide a description of the build lifecycle.

An example travis.yaml file is provided below:

```
sudo: true
dist: trusty
language: go
```


```
go:
- 1.7.x
- 1.8.x
addons:
  apt:
    packages:
    - libvirt-bin
    - libvirt-dev
env:
  global:
  - ORG_PATH=/home/travis/gopath/src/github.com/intelsdi-x
  -
SNAP_PLUGIN_SOURCE=/home/travis/gopath/src/github.com/${TRAVIS_REPO_SLUG}
  - GLIDE_HOME="${HOME}/.glide"
  matrix:
  - TEST_TYPE=small
  - TEST_TYPE=build
matrix:
  exclude:
  - go: 1.7.x
    env: TEST_TYPE=build
before_install:
- "[[ -d $SNAP_PLUGIN_SOURCE ]] || mkdir -p $ORG_PATH && ln -s
$TRAVIS_BUILD_DIR $SNAP_PLUGIN_SOURCE"
install:
- cd $SNAP_PLUGIN_SOURCE
- make deps
script:
- make check 2>&1
```

The automated continuous integration tests managed by Travis CI for the Snap libvirt Collector Plugin developed by MIKELANGELO is illustrated in Figure 7.

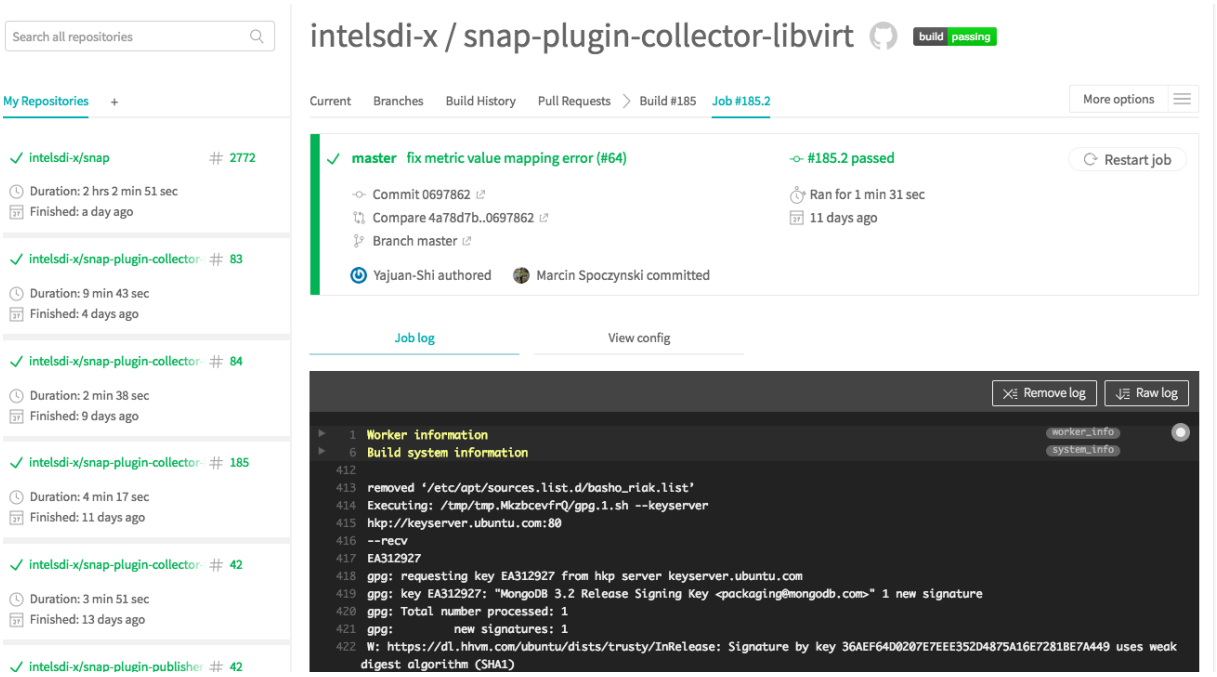


Figure 7: Automated Continuous Integration tests managed by Travis CI

Travis is also used for code reviews to check that a new feature or bug fix will not break the existing project codebase. When one of the project contributors creates a pull request, a Travis unit test job is scheduled and a reviewer, usually the owner of the plugin/repository, needs to wait for Travis feedback before code can be merged. With each commit, Github allows a clean interface for general comments as well as specific comments on a line of code as illustrated in Figure 8.

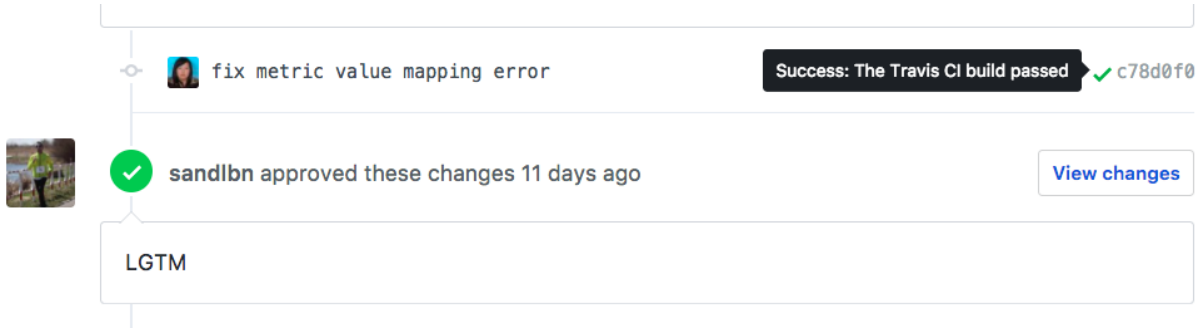


Figure 8: Automated Continuous Integration tests managed by Travis CI integrated with Git Review process

The ability to raise comments or questions on every single line of code is very useful in doing line by line code reviews. When changes are reviewed and approved, the Github review bot calls the Travis CI API and it schedules a build. If the build is successful Travis releases a new

version of the plugin and publishes results to the Slack and Github release page as illustrated in Figure 9.

Figure 9: Github Libvirt plugin download page managed by Travis CI

Integration tests exercise the interactions between different components of a system. As such, integration tests often require a testing database like InfluxDB or virtual infrastructure components like kvm or libvirt. We decided to set up an automated integration testing environment so that we can be confident that the components of our system continue to communicate correctly with each commit. For Integration tests need to install and exercise components. We decided to use Jenkins which is the most popular CI platform, allowing us to set it up and use it in minutes on our local infrastructure. Instead of deploying Jenkins on public cloud infrastructure, we decided to use a bare metal installation and use a private openstack cloud as CI executors. For the configuration we are using the same travis.yaml file that is used for unit testing. To build that pipeline and integrate the travis configurator description we used the travis-yaml parser https://github.com/travis-ci/travis-yaml, along with a custom built library to map configuration to the Jenkins executors. For every plugin a job is triggered daily, and also when a new version of the plugin is published.

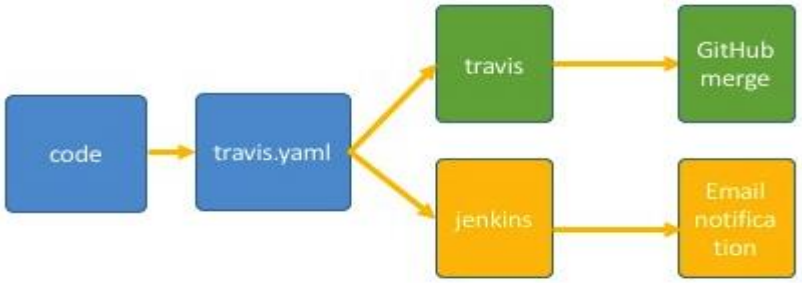


Figure 10: Standard workflow for Snap plugin development cycle with Travis CI and Jenkins together

3.2 Lightweight Execution Environment Toolbox

Until now we focused mainly on integrating the MIKELANGELO stack into **VM-based** cloud platforms such as OpenStack (see D5.2, 3.2.1). While such integrations work, VM-based platforms turned out not to be well suited for running lightweight OSv unikernels. While such platforms excel at managing large isolated VMs, they lack means of automatically provisioning, maintaining and interconnecting a number of tiny VMs. OpenStack, for example, doesn't even support specifying flavor with HDD size smaller than 1GB resulting in resource quotas being thrown away when tiny VMs (50MB or so) are provisioned. Furthermore, there is no obvious way of managing large number of VMs originating from the same virtual machine image, i.e. replicas, to support microservice architecture design where a malfunctioning instance is expected to get replaced by a fresh replica automatically. While VM-based platforms excel at connecting isolated VMs to the network, they are weak at automatically interconnecting replicas with other VMs.

On the other hand **container-based** cloud platforms are meant to run numerous interconnected lightweight workloads by design. Demand-based scaling and self-healing of those parts of the application that are experiencing difficulties can be built into the management platform and controlled automatically.

In the following subsections we first briefly outline our contributions from the first two project years. Then we continue with a short introduction of the Kubernetes platform and its pluggable runtime Virtlet that enables QEMU/KVM virtual machines deployment on Kubernetes side-by-side with Docker containers. We conclude by presenting three real example deployments to demonstrate MIKELANGELO stack integration into the Kubernetes platform. All of these enhancements contribute to the Lightweight Execution Environment Toolbox (LEET).

3.2.1 Overview of Past Contributions

As briefly mentioned in the introduction to this section, management of lightweight applications previously focused primarily on standard cloud environments, such as OpenStack and Amazon. This section therefore briefly presents achievements that made use of unikernels in these environment simpler.

The first iteration of enhancements was integrated directly into Capstan, a tool used for managing application packages and composing unikernels from these packages. We designed a provider abstraction and implemented it for OpenStack. This provider is straightforward and supports only two basic operations: publishing OSv images into OpenStack and provisioning them. The tool has been extended to resemble local execution as far as possible, using the same command line switches and completely reusing the

approach for image composition. The major difference is related to the user authentication and authorisation. Whereas a local environment assumes the user running Capstan tool is the owner of the local package and image repository, the OpenStack provider needs to authenticate the user against the provided backend API (Keystone - Identity service[11]).

While these capabilities are essential for the integration with the cloud provider, difficulties experimenting with various workflows have been noticed. Once the instances are provisioned, there is no way to query their status, restart or shut them down. In order to accomplish these tasks, users are required to access the cloud either through the dashboard or by using the corresponding cloud API manually.

This led to a research of alternate technologies that could be used to manage more complete life cycles of these unikernels. Around this time a new open source project, UniK[12], was released by EMC[13]. Based on the speed of development and rapid adoption by the community it appeared like a reasonable basis for our next enhancements. Our contributions added two major achievements: integration of MIKELANGELO version of Capstan with the full support for application packages and implementation of the OpenStack provider. Altogether, this allowed users of UniK to deploy any kind of unikernel (not just OSv, also IncludeOS and Rumprun) in OpenStack. To MIKELANGELO, on the other hand, this integration enabled us to deploy our application, such as OpenFOAM Cloud, in Amazon Web Services and Google Compute Engine.

Unfortunately, in Q2 of 2017 the UniK project lost momentum, when the core team left EMC.

3.2.2 Short Introduction to Kubernetes

Kubernetes is a portable self-healing platform meant to run Docker containers. It's designed with scalability and container interconnectivity in mind, which reflects on its basic concepts:

Pod. Pod is the smallest unit one can deploy on Kubernetes. In most cases a Pod includes a single container, but auxiliary containers are also allowed to support the main container. Together they represent one running process. Each Pod is assigned one or more key-value pairs (labels) and a single unique IP address. When a Pod dies it cannot be resurrected, but rather replaced with equivalent fresh Pod, i.e. replica, that is assigned the same labels but a new IP address.

Service. Service is an addressable abstraction that allows us to address one or more Pod replicas filtered by labels. One can think of it as a load-balancer over equivalent Pods.

Deployment. Deployment is a declarative controller for Pod replicas. One specifies a desired state, e.g. number of Pod replicas to be deployed, and the controller will execute it.

While Kubernetes is originally meant to manage Docker containers, its CRI[14] (Container Runtime Interface) allows for arbitrary lightweight workloads to be run. Early in 2017 a revolutionary open-source CRI implementation was presented that allows for QEMU/KVM workloads to run on Kubernetes - Virtlet[15].

3.2.3 Virtlet - Virtualisation on Kubernetes

Virtlet is a CRI implementation that allows QEMU/KVM powered workloads based on QCOW2 images to be run on Kubernetes. It is being developed by Mirantis. Similarly to other components MIKELANGELO builds upon, Virtlet is an open source project hosted on Github.

What Kubernetes names a "Pod" is considered a "QEMU/KVM workload" (VM) in Virtlet, leaving all other entities intact. The VM's networking is managed by Virtlet so that interconnectivity between Pods is unmodified compared to plain Kubernetes.

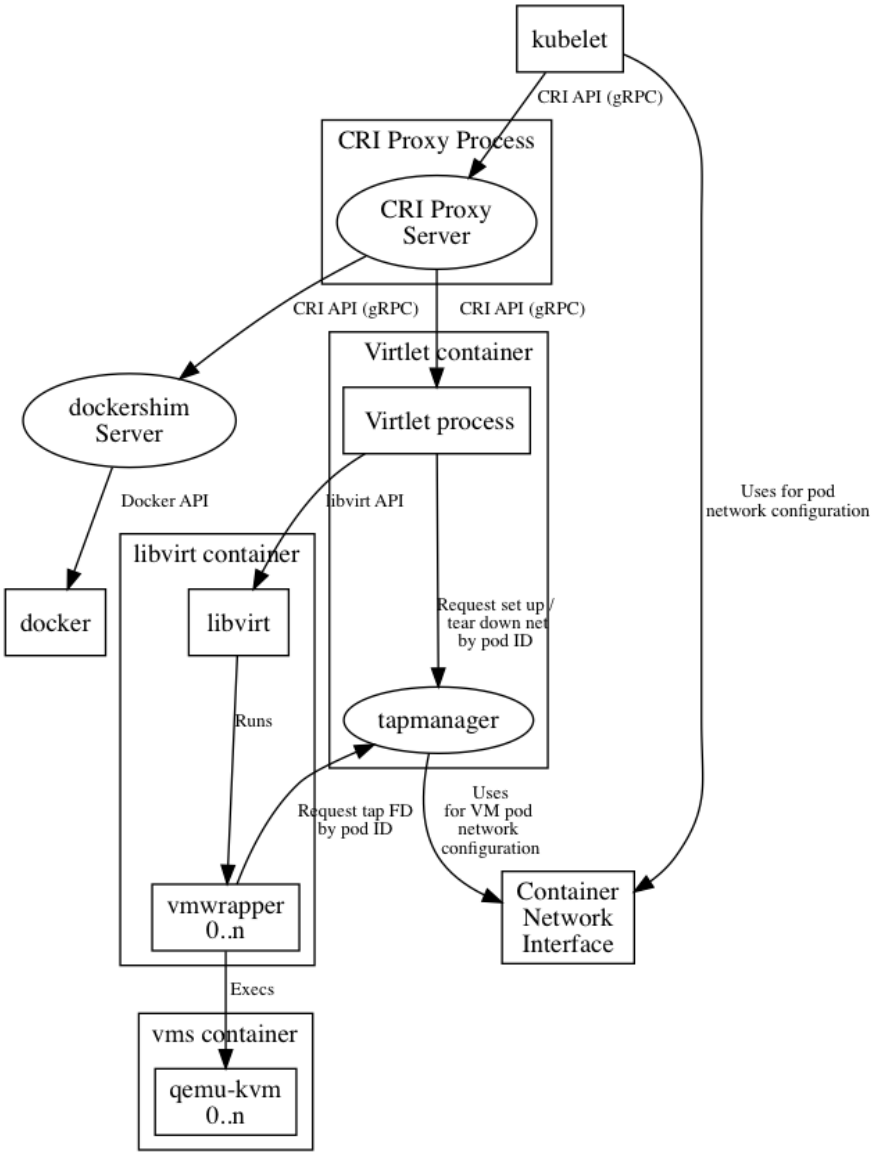


Figure 11: Virtlet architecture.

Virtlet is deployed to Kubernetes as a Pod consisting of three Docker containers: (i) *Virtlet container* that intercepts CRI requests and manages the VM networking. It also delegates actual VM provisioning request to libvirt daemon that runs in (ii) *Libvirt container* which initializes the qemu-kvm processes inside (iii) *VMs container*, as depicted in Figure 11[16]. Virtlet Pod must be deployed on each node of the Kubernetes cluster that is to support deployment of VMs.

Having Virtlet Pod deployed on Kubernetes, the user can deploy both Docker-based Pods or QEMU/KVM-based Pods using the same Kubernetes deployment YAML specification. The following listing shows a minimalistic example of a valid Pod specification for a QEMU/KVM-based Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: cirros-vm
  annotations:
    kubernetes.io/target-runtime: virtlet.cloud
spec:
  containers:
    - name: cirros-vm
      image: virtlet.cloud/download.cirros-cloud.net/0.3.5/cirros-
                                                0.3.5-x86_64-disk.img
```

As shown in the above example, this YAML configuration is fully compatible with standard Kubernetes configuration. Virtlet is using existing settings to allow for specification of the VM. The only difference is that QEMU/KVM-based Pods must adhere to the following criteria:

- The Pod must include an annotation for `kubernetes.io/target-runtime` with value `virtlet.cloud`
- The Container image must always be prefixed with `virtlet.cloud/`
- Only one container configuration can be specified per Pod

3.2.3.1 MIKELANGELO Contributions

Part of the WP5 effort was focused on integrating the MIKELANGELO stack into Kubernetes. OSv unikernels depend on QEMU/KVM support, therefore the Virtlet CRI implementation had to be used. With the Virtlet codebase being relatively new and immature, OSv unikernels weren't able to run smoothly on Kubernetes out of the box, therefore some MIKELANGELO effort was spent to ensure that required features were supported.

Logging. The first integration attempt revealed that Virtlet had no support at all for accessing VM logs i.e. it was impossible to view what application has sent to standard output and standard error. At the time being, the desired workflow to debug potential application issues was by connecting to the VM (running on Kubernetes) by means of `virsh` or `ssh`. With OSv unikernels such approach was not possible by definition due to the absence of both an interactive shell and a ssh daemon, therefore the ability to access VM logs was crucial. Since Virtlet core developers had no plans to support it in a time suitable for the MIKELANGELO project we had to design, implement and test it on our own. Figure 12 shows the logging architecture as we have ultimately implemented it. Our solution included redirecting VM stdout/stderr into a file by configuring libvirt properly. A Go program was then implemented to convert the file into a form understandable by Kubernetes. We've added an additional container (mirantis/virtlet-log) to the Virtlet Pod that was responsible for that. As a result, VM logs are now available from within Kubernetes Dashboard in the same

way as they are for Docker-based Pods. Our changes were merged into upstream and used as part of several demonstrations held by the Virtlet team.

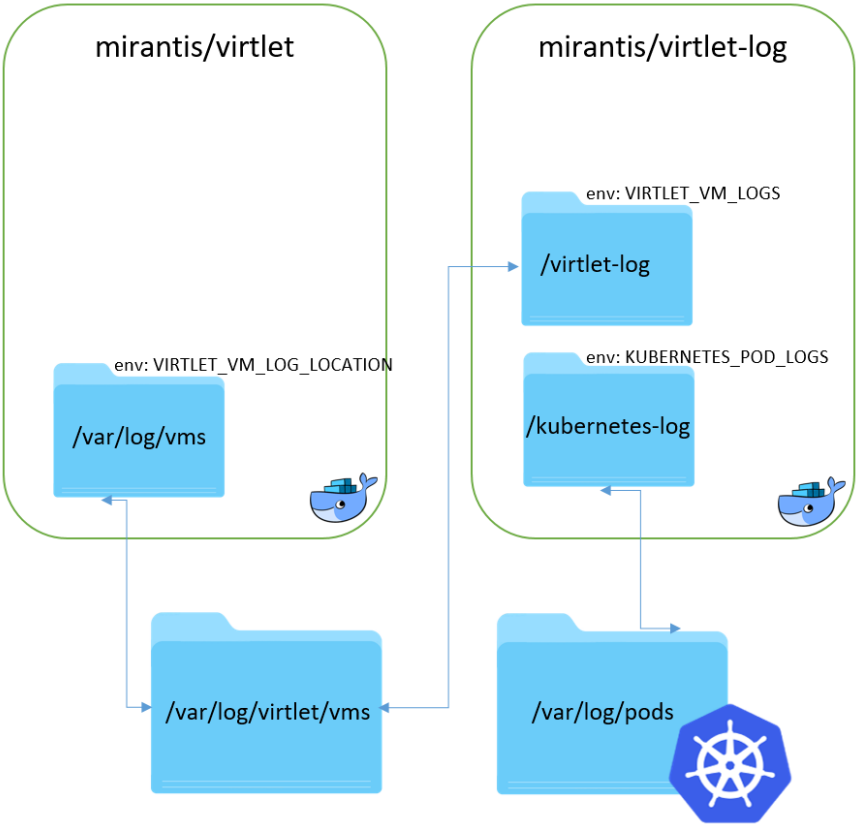


Figure 12: Architecture of logging mechanism as implemented under MIKELANGELO project.

Minor Enhancements & Testing. As the first real-world user of the Virtlet project we were subject to numerous bugs that Virtlet developers didn't foresee. Also, due to Virtlet being in immature beta stage, there were some breakable changes published now and then that enforced us to react either by reporting an issue or by providing the actual solution prior to continuing with MIKELANGELO stack integration. There were 17 issues reported by us and 6 actual solutions provided by us at the time of writing.

3.2.4 Integrated Capabilities

The MIKELANGELO stack is now satisfyingly well integrated into Kubernetes, where multiple aspects have been tested. Below we list the most crucial aspects of integration that were tested and are also demonstrated in the next chapter with example applications.

Baked-in boot command. The initial aspect that was tested was whether OSv unikernel is able to run on Kubernetes at all i.e. without any specific configuration. A simple Node.js unikernel with baked-in boot command was prepared by means of LEET toolbox and

deployed to Kubernetes. Nothing worked out of the box and we had serious troubles debugging it due to the absence of VM logs. Having logging functionality implemented by us we were finally able to successfully provision the unikernel, but were facing notably poor virtualization performance. After configuring Virtlet to use KVM support instead of plain QEMU, it eventually worked well, the unikernel was properly provisioned and the Node.js application started.

Cloud-init. An OSv unikernel was prepared with LEET toolbox containing osv.cloud-init package with boot command set to `/bin/sleep.so -1`, effectively sleeping forever. The sleep.so utility program is part of implicitly required osv.bootstrap package and was implemented as part of Kubernetes integration. The unikernel was then uploaded to Kubernetes with the following additional annotations:

```
annotations:
  VirtletDiskDriver: virtio
  VirtletCloudInitUserDataScript: |
    run:
      - PUT: /app/
        command: "runscript /run/app"
```

A NoCloud cloud-init virtual CD-ROM is attached to the unikernel with user data content taken from `VirtletCloudInitUserDataScript` annotation. By default it gets attached as a virtio-scsi CD-ROM which is currently not supported by OSv, therefore the `VirtletDiskDriver` annotation must be used to request the CD-ROM device being attached as a virtio-blk devices. With the two annotations provided, unikernel is able to access cloud-init data and interpret it. When it became evident that unikernel was able to consume cloud-init data properly an extra effort was spent to simplify boot commands as far as possible for the cloud-init data to remain as readable as possible, as described in report D4.6 (chapter 5.1, bullet 3).

Volumes. Attaching a QCOW2 flexvolume (the volume that is created upon unikernel creation and destroyed upon its termination) to the unikernel was tested with a unikernel that had the following boot command set:

```
runscript /run/format_volume1; runscript /run/app
```

There are two parts to it: the first part (left of the semicolon) ZFS-formats the QCOW2 volume that is attached to /dev/vblk1 and mounts it into /volume directory. The `format_volume1` run configuration is provided by the implicitly required osv.bootstrap package to prevent the user from using the complex zpool command directly and was implemented as a part of Kubernetes integration. The second part (right of the semicolon) eventually runs a simple application that uses the volume by writing and reading files in the mounted directory. The

following volume specification has to be added to the Pod specification for Virtlet to create and attach a flex volume to the unikernel during the provisioning:

```
volumes:
  - name: volume1
    flexVolume:
      driver: "virtlet/flexvolume_driver"
      options:
        type: qcow2
        capacity: 512MB
```

Interconnectivity. Unikernel interconnectivity was tested to validate that unikernels are able to communicate among each other and with regular Docker-based Pods. A microservice demo application (see Section 3.2.5.3) was designed where we test whether OSv unikernels are able to properly consume Kubernetes Services. Unfortunately it didn't work initially, but by reporting a descriptive issue with a repeatable example to the Virtlet core developers they were able to fix a bug in Virtlet networking for us and QEMU/KVM workloads, such as OSv unikernels, were then able to consume Services normally.

Replicas. After assuring that a single replica of OSv unikernel (a Pod) runs smoothly on Kubernetes, we were able to proceed to testing Kubernetes Deployments that manage more than one replica of the same unikernel. Networking issues were detected when many replicas were to be provisioned in a short period of time. An issue was reported to Virtlet core team and fixed eventually so that provisioning multiple replicas at once or scaling them up or down now works normally.

Debugging. An important aspect of Virtlet integration is the possibility to debug when a unikernel-driven application doesn't work. Having the logging functionality implemented that lets the user examine the unikernel's stdout/stderr from the Kubernetes Dashboard, we conclude that the debugging functionality is now at a sufficient level.

3.2.5 Real-World Demonstrator Applications

In this section we present three working applications that demonstrate MIKELANGELO stack integration into Kubernetes. All applications are open sourced and available on the MIKELANGELO GitHub portal.

3.2.5.1 OpenFOAM on Kubernetes

OpenFOAM serves as the basis of the MIKELANGELO Aerodynamics use case. It offers a large set of tools for running complex CFD (Computational Fluid Dynamics) simulations. The use case has experimented with different deployment scenarios, from a dedicated OpenStack application, to vTorque compliant set of scripts. In this section we describe how a complete

OpenFOAM simulation with four threads (for a simple example) running in two OSv unikernel replicas can be configured to be deployed on a Kubernetes cluster. As can be seen from Figure 13 our setup mixes both Docker containers and OSv unikernels that communicate with each other via Kubernetes Services.

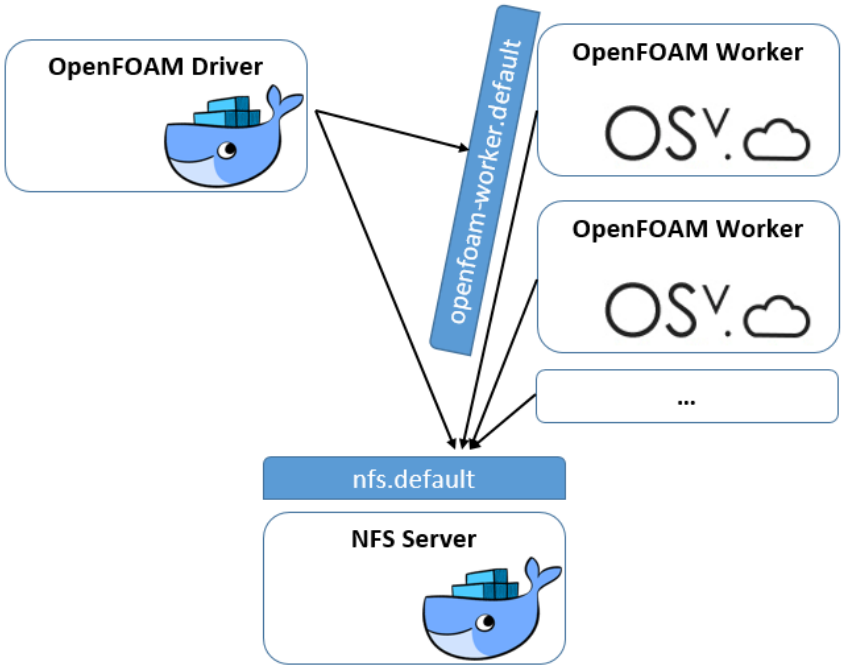


Figure 13: OpenFOAM example application setup on Kubernetes.

There are two Docker containers (OpenFOAM Driver and NFS Server) deployed together with an arbitrary number of OSv unikernel replicas (in our case there were two replicas deployed). Kubernetes Services are drawn in blue - one in front of the NFS Server and one in front of OpenFOAM Worker replicas.

The complete setup is available in mikelangelo-project/osv-openfoam-demo GitHub[17] repository and can be deployed with a single command (assuming that a Kubernetes cluster is up-and-running and has Virtlet runtime installed):

```
$ kubectl create -f virtlet_deploy/openfoam.yaml
```

The command above will deploy the following components to the Kubernetes cluster:

- "nfs" Deployment (with a single replica)
- "nfs.default" Service
- "openfoam-driver" Deployment (with a single replica)
- "openfoam-worker" Deployment (with two replicas)
- "openfoam-worker.default" Service

NFS. An unmodified freely available third-party Network Filesystem (NFS) Docker image is used to host a shared directory /case that both OpenFOAM Driver and OpenFOAM Workers have access to. The NFS is proxied with a Service to make it addressable as "nfs.default" from within the Kubernetes cluster so that other workloads (Driver and Workers) can address it using the immutable name, rather than their mutable IP addresses.

OpenFOAM Driver. A RHEL 6.6 Docker image with preinstalled OpenFOAM distribution is used as a base for mikelangelo/openfoam-driver image and then some additional utility software and scripts are added to support the demo. The purpose of this container is to (i) prepare simulation data i.e. to download the case, decompose it and upload it to the NFS, (ii) initialize simulation on OSv workers and (iii) to reconstruct the partial results obtained from the workers when they are done calculating. Total size of the container is 580 MB.

OpenFOAM Worker. An OSv unikernel image with the three packages osv.cloud-init, osv.nfs and openfoam.simplefoam-2.4.0 is used and the boot command set to sleep forever. Total size of the unikernel image is 55 MB. Unikernel is given a command to mount NFS via Virtlet's cloud-init mechanism when provisioned and then it waits for Driver to initialize OpenFOAM simulation for it, as can be seen in Worker's Deployment specification:

```
annotations:
  kubernetes.io/target-runtime: virtlet.cloud
  VirtletDiskDriver: virtio
  VirtletVCPUCount: "2"
  VirtletCloudInitUserDataScript: |
    Run:
      - PUT: /app/
        command: "runscript /run/mount-case"
      - PUT: /app/
        command: "/bin/echo.so Worker started with NFS mounted
                                                    to /case."
```

There can be any number of Worker replicas deployed with each of them having 2 VCPUs allocated - and therefore being able to run two OpenFOAM threads in parallel. Worker replicas are labelled with "case: openfoam-worker" and proxied with a Service to make them addressable as "openfoam-worker.default" from the Driver. Below please find a full Worker's Service specification:

```
apiVersion: v1
kind: Service
metadata:
  name: openfoam-worker
  namespace: default
```

```
spec:
  type: ClusterIP
  selector:
    case: openfoam-worker
  ports:
    - name: httpserver
      port: 8000
```

Having such a Service deployed, Driver is then able to initialize simulation by sending a HTTP request to http://openfoam-worker.default:8000/app/ effectively hitting a randomly picked replica of the OSv Workers.

To run the demo one needs to execute the following command on their host computer (assuming that containers and unikernels are already deployed):

```
$ ./demo.sh
```

The script will remotely execute a sequence of scripts inside Driver container. First, NFS will get mounted to Driver's /case folder and example simulation data will be downloaded from S3 repository. Then it will decompose the example data and initialize the simulation on Workers. Finally, the partial results will get reconstructed inside the Driver container. When the script is done, the simulation result is available inside the /case NFS directory.

3.2.5.2 Apache Spark on Kubernetes

In this section we describe how a complete Apache Spark application can be deployed on Kubernetes. Figure 14 illustrates the deployment: it contains a single Docker container (Spark Driver) and two kinds of OSv unikernels, the Spark Master and the Spark Workers. The complete setup is available in mikelangelo-project/osv-spark-demo GitHub[18] repository and can be deployed with a single command (assuming that Kubernetes cluster is up-and-running and has Virtlet runtime installed):

```
$ kubectl create -f virtlet_deploy/spark.yaml
```

The command above will deploy the following components to a Kubernetes cluster:

- "spark-master" Deployment (with a single replica)
- "spark-master.default" Service
- "spark-worker" Deployment (with two replicas)
- "spark-driver" Deployment (with single replica)

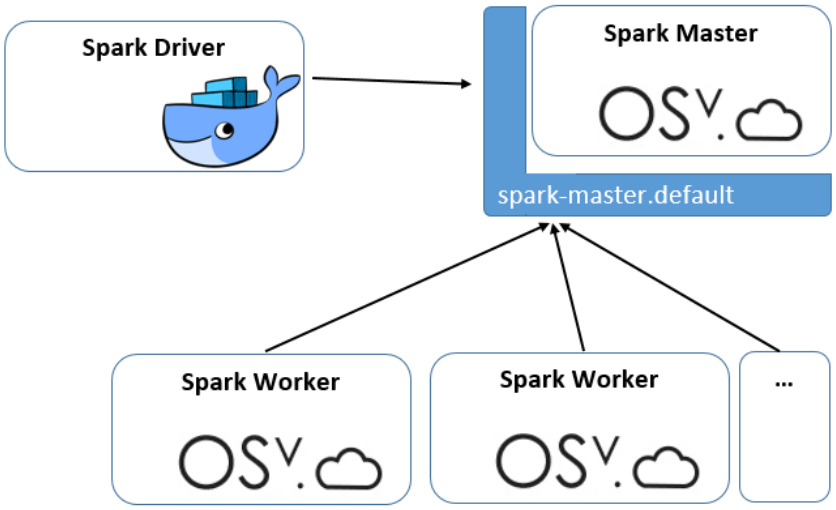


Figure 14: Apache Spark deployment on Kubernetes.

Spark Master. An OSv unikernel image with the two packages osv.cloud-init and apache.spark-2.1.1 is used and the boot command set to sleep forever. Total size of the unikernel image is 192 MB. Unikernel is given a command to run Spark Master via Virtlet's cloud-init mechanism when provisioned, as can be seen in its Deployment specification:

```
annotations:
  kubernetes.io/target-runtime: virtlet.cloud
  VirtletDiskDriver: virtio
  VirtletCloudInitUserDataScript: |
    run:
      - POST: /env/PORT
        val: 7077
      - POST: /env/UIPORT
        val: 8888
      - PUT: /app/
        command: "runscript /run/master"
```

The cloud-init script first sets two environment variables, PORT and UIPORT, that are then consumed by /run/master run configuration. The master run configuration is defined within apache.spark-2.1.1 package as:

```
runtime: native
config_set:
  master:
    bootcmd: >
      /java.so
      -Xms$XMS
      -Xmx$XMX
```

```
      -cp /spark/conf:/spark/jars/*
      -Dscala.usejavacp=true
      org.apache.spark.deploy.master.Master
      --host $HOST
      --port $PORT
      --webui-port $UIPORT
    env:
      XMS: 512m
      XMX: 512m
      HOST: 0.0.0.0
      PORT: 7077
      UIPORT: 8080
```

One can notice how the user actually doesn't need to provide the exact command in the cloud-init, but can make use of pre-prepared run configurations that are provided within packages. By specifying `PORT` and `UIPORT` environment variables in the cloud-init above we only customize the bit that we need to customize.

Spark Master is proxied with a Service to make it addressable as "spark-master.default" from within Kubernetes cluster so that other workloads (Workers and Driver) don't deal with mutable IP address.

Spark Worker. An OSv unikernel image equal to that of a Spark Master is used to run Spark Worker, only cloud-init invokes a different run configuration in it, as can be seen in its Deployment yaml specification:

```
annotations:
  kubernetes.io/target-runtime: virtlet.cloud
  VirtletDiskDriver: virtio
  VirtletCloudInitUserDataScript: |
    run:
      - POST: /env/MASTER
        val: spark-master.default:7077
      - PUT: /app/
        command: "runscript /run/worker"
```

An environment variable MASTER is first set to "spark-master.default:7077" and is then consumed by the worker's runscript that runs the Apache Spark worker thread which automatically registers itself to the Master address. Two Worker replicas are deployed by default, but the number can easily be increased at any time.

Spark Driver. A third-party Debian based Docker image with preinstalled Spark & Hadoop distribution is used as a base for the mikelangelo/spark-driver image and then some

additional utility software is added to support the demo. The sole purpose of this container is to submit the Spark job to the Spark Master. Total size of the container is 460 MB.

To run the demo one needs to execute the following command on their host computer (assuming that containers and unikernels are already deployed):

```
$ ./demo.sh
```

The script will remotely execute a Spark submit command inside the Driver container. A demo application will be submitted that runs 1000 tasks to calculate PI decimals. The calculation result, i.e. PI estimate, is printed to the console.

3.2.5.3 Microservices Demo on Kubernetes

A synthetic microservice application was implemented as part of the MIKELANGELO project to demonstrate how OSv unikernels can be used to support modern microservice architectures. A detailed description of the application is described on a MIKELANGELO blog post[19], therefore we only pinpoint the most interesting bits here to demonstrate how it can be deployed on Kubernetes.

Figure 15 shows the application components, all of them deployable with a single command (assuming that Kubernetes cluster is up-and-running and has Virtlet runtime installed):

```
$ kubectl create -f virtlet_deploy/micro.yaml
```

The command above will deploy the following components to a Kubernetes cluster:

- "micro-master" Deployment with a single replica (see *Master* on Figure 15)
- "micro-master.default" Service
- "micro-storage" Deployment with a single replica (see *Task store* on Figure 15)
- "micro-storage.default" Service
- "micro-keyvaluestore" Deployment with a single replica (see *Key-value store* on Fig. 15)
- "micro-keyvaluestore.default" Service
- "micro-db" Deployment with a single replica (see *File storage* on Figure 15)
- "micro-db.default" Service
- "micro-worker" Deployment with two replicas (see *Workers* on Figure 15)
- "micro-ui" Deployment with a single replica (see *Frontend* on Figure 15)

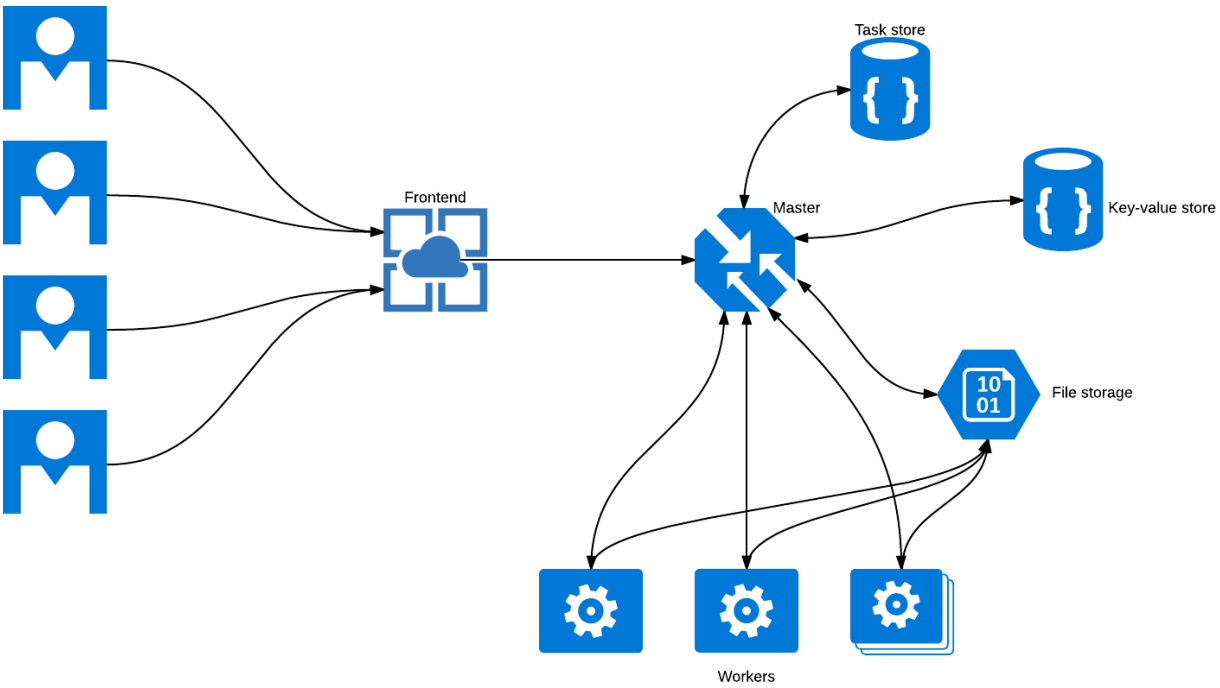


Figure 15: Microservice architecture.

OSv unikernels. A single OSv unikernel image with the two packages osv.cloud-init and node-4.4.5 is used with boot command set to sleep forever. Total size of the unikernel image is 45 MB. All six components of the application (Master, Task store, Key-value store, File storage, Workers, Frontend) use this very same image, but with a different boot command provided by means of cloud-init. The manifest below shows available run configurations, all using Node.JS as their base runtime type:

```
runtime: node
config_set:
  master:
    main: /master.js
  storage:
    main: /storage.js
  keyvaluestore:
    main: /keyvaluestore.js
  db:
    main: /db.js
  worker:
    main: /worker.js
  ui:
    main: ui.js
```

The above run configuration is transformed into six corresponding files when building a unikernel with LEET toolbox, all six of them being executable by the `runscript` program which is included in every OSv unikernel by default:

- /run/master
- /run/storage
- /run/keyvaluestore
- /run/db
- /run/worker
- /run/ui

Kubernetes Deployment definitions. Each OSv unikernel is deployed to Kubernetes as a QEMU/KVM-based Pod whose replicas are managed by Deployment specification, as can be seen in Deployment's yaml specification:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: micro-worker
spec:
  replicas: 2
  template:
    metadata:
      name: micro-worker
      annotations:
        kubernetes.io/target-runtime: virtlet.cloud
        VirtletDiskDriver: virtio
        VirtletCloudInitUserDataScript: |
          run:
            - POST: /env/PORT
              val: 9000
            - POST: /env/MICRO_KEYVALUESTORE_ENDPOINT
              val: micro-keyvaluestore.default.svc.cluster.local
            - PUT: /app/
              command: "runscript /run/worker"
    spec:
      containers:
        - name: micro-worker
          image: virtlet.cloud/s3.amazonaws.com/osv-microservice-
                                                  demo/micro.qemu
```

The YAML specification above deploys two replicas of Worker to the Kubernetes, each replica being an independent OSv unikernel based on micro.qemu image that was obtained from a publically available URL (S3 repository in this case). The cloud-init script is comprised of three

commands that are run inside the unikernel when it's booted. Initially, the PORT and `MICRO_KEYVALUESTORE_ENDPOINT` environment variables are set which are later consumed by the Worker application. Then the `/run/worker` script is executed effectively starting the Worker. A similar approach is used to boot other unikernels as well, where only the cloud-init script is adjusted.

The above three examples of realistic applications show the potential of mixing OSv-based unikernels built with the support of MIKELANGELO LEET with other lightweight environments, such as containers. This mix and match approach finally allows end users to choose the best tool and the best environment for the task at hand. Because common management platform is employed, users need not to learn new ways of managing clustered applications, reducing the costs of adopting these new technologies.

3.3 Scotty

Scotty is our year 3 implementation of the continuous experimentation concept. Scotty uses concepts and components from the field of continuous integration and applies them to run infrastructure experiments. In this section we first discuss the motivation behind Scotty's development. Then we present the past approaches, which were implemented in the first two project years. Based on the experiences from the first two years, we then present the current architecture for Scotty. We then mention how Scotty integrates with cloud middleware and other MIKELANGELO components. We conclude this section with an outlook of future features in Scotty, to be developed beyond the project.

3.3.1 Motivation

The motivations to develop Scotty revolve around running automated infrastructure experiments in the cloud and HPC. The requirements for Scotty span the needs to run complex infrastructure experiments, run experiments repeatedly, run reproducible experiments, having an audit trail, providing streamlined workflows to run experiments, reusability of workloads, and getting rid of boilerplate code.

Running complex infrastructure experiments is important in the context of MIKELANGELO, because the developed components span a relatively large area of the software infrastructure of a datacenter.

Repeating experiments is required to assess the variability of experimental configurations. Only by repeating experiments many times over, does it become possible to assess the variability and subsequently compare configurations based on many runs statistically.

Reproducibility of experiments is required as part of scientific best practice. Strikingly good and strikingly bad results should be reproducible to prove their merit or to analyze problems in configurations.

An audit trail of performed experiments supports the scientific process. Without an audit trail scientists may tend to only publish positive results, without disclosing failed experiments. Thus, a bias to good results can be established, which in turn undermines the positive results in the first place.

Streamlined workflows are important for adoption. Having an automated experimental system does not help anyone if setting up and running experiments takes longer with the system than without. The barriers are expected to be even higher. Since the researchers initially might not trust the system's functionality and its benefits, they might still decide to perform experiments manually. Thus, the ease with which the system can be used needs to be compelling.

Experiments run workloads, which in turn should be reusable. If experiments need to redevelop the same workloads, such as a database benchmarking workload, individually, the burden to run experiments is still large. On the other hand, being able to reuse workloads developed by others opens up the possibility to define and run new experiments very quickly.

Finally, another large possibility for Scotty to resolve inefficiencies in running infrastructure experiments lies in getting rid of boilerplate code and configuration. Boilerplate code consists of typical processes and infrastructure that are needed by most experiments. These processes consist of metering, collecting static data, extracting data for analysis, setting the infrastructure configuration, deploying and starting workloads, and publishing experimental data.

3.3.2 Past Approaches

The development of running automated experiments went through three main phases. These phases correspond to the three project years. The phases went from manual experimentation, via a complex CI setup, to today's implementation with Scotty.

In the first project year experiments have been performed manually. For example, in the big data use case the big data cluster has been set up on a set of physical hosts from scratch. Then HiBench[20] was installed and run on the cluster. The overall approach took 3 months to finish and collect data. The initial experimentation has shown that automation will be needed for future experiments. So, at the end of the first project year, we have set out to automate experimentation with Jenkins. Jenkins was used to start experiments. However, soon we realized that the main bottlenecks lie in the deployment of infrastructure and

experiments. This realization led us to the development of a more automated system in the second project year.

In the second project year, we have deployed a testing and integration system based on OpenStack's approach. The approach includes Gerrit, Zuul, Jenkins and a few extensions. Gerrit is a code management and review system developed by Google. In our setup Gerrit served to host code repositories that define the experiments. Zuul is a pipeline that allows the assembly of experiments with specific code that should be integrated in the experiment. Jenkins is a CI platform that manages the execution of experiment jobs. As part of the experiment execution, Jenkins managed the provisioning of resources in OpenStack and HPC with a tool called nodepool. The overall setup consisted mostly of third-party components. However, custom configuration and custom logic was necessary to enable the experimentation workflow.

Figure 16 shows the overall approach to run experiments in the experimentation framework in the second project year. First, the researcher would push an experiment definition as a YAML file to a repository in Gerrit. Once Gerrit would receive and verify the experiment, Zuul would collect configuration and code to perform the experiment and trigger Jenkins. Jenkins would then first provision the resources needed to perform the experiment. Jenkins was connected to node pool, which in turn provided resources, such as VMs, via OpenStack. Once node pool would provide the required resources, Jenkins would execute the experiment. During the experiment data was collected with Snap and written to result stores, such as InfluxDb and MongoDB.

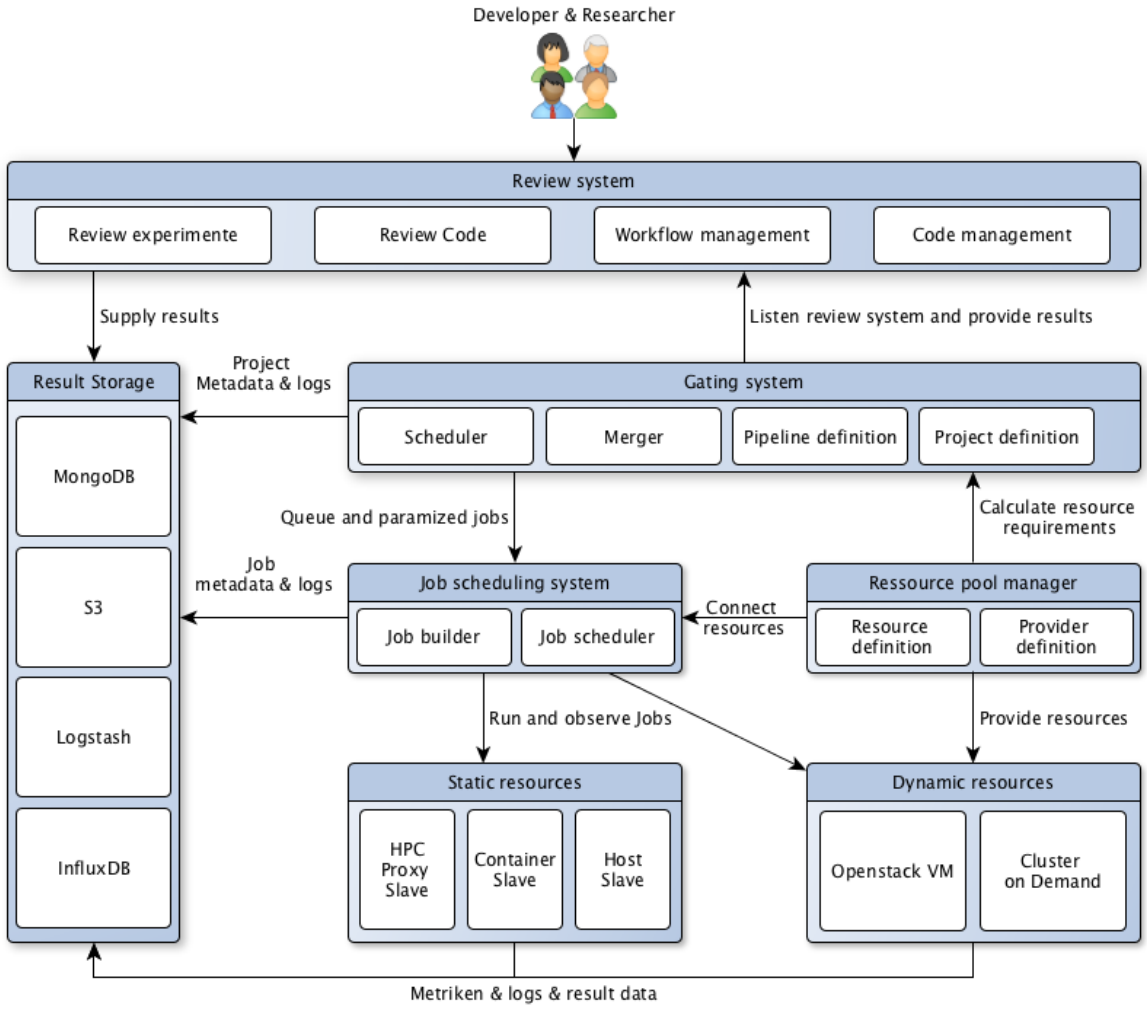


Figure 16: Architecture of Scotty's year 2 prototype.

The solution in the second project year worked sufficiently well for a constrained set of scenarios. This solution has been used for the data analytics use case, for the OpenFOAM use case, and for the cancellous bones use case. However, the approach in the second project year was fragile, prone to errors, and hard to extend. Due to the complexity of the system there were regular problems due to the integration of the assembled components. Changes to configuration required a restart of most parts of the system. The layers of abstraction between experiments and resources posed a problem when defining the use of those experiments. Another problem arose from the fact that more and more custom logic was required to implement the experiment workflow. However, the logic was scattered in different types of configuration files (JSON, YAML, and others) and different types of code files (mostly Python and Bash) throughout various components (Gerrit, Zuul, Jenkins, node pool). At the same time it turned out that we would not require many of the more complex features of Gerrit, Zuul, and Jenkin. Thus, in year three we have decided to build a new system

from scratch based on our experience from the first two years.

3.3.3 Architecture

The current implementation is based on a redesign, based on the experiences made with the first two versions of continuous experimentation. The new incarnation of the concept is called Scotty. A main consideration when implementing Scotty was the consolidation of configuration and code in one place. Thus, Scotty is one centralized python package, which integrates with several components from its ecosphere. This approach allows for easier development, testing, and deployment of Scotty. Figure 17 shows Scotty and its environment.

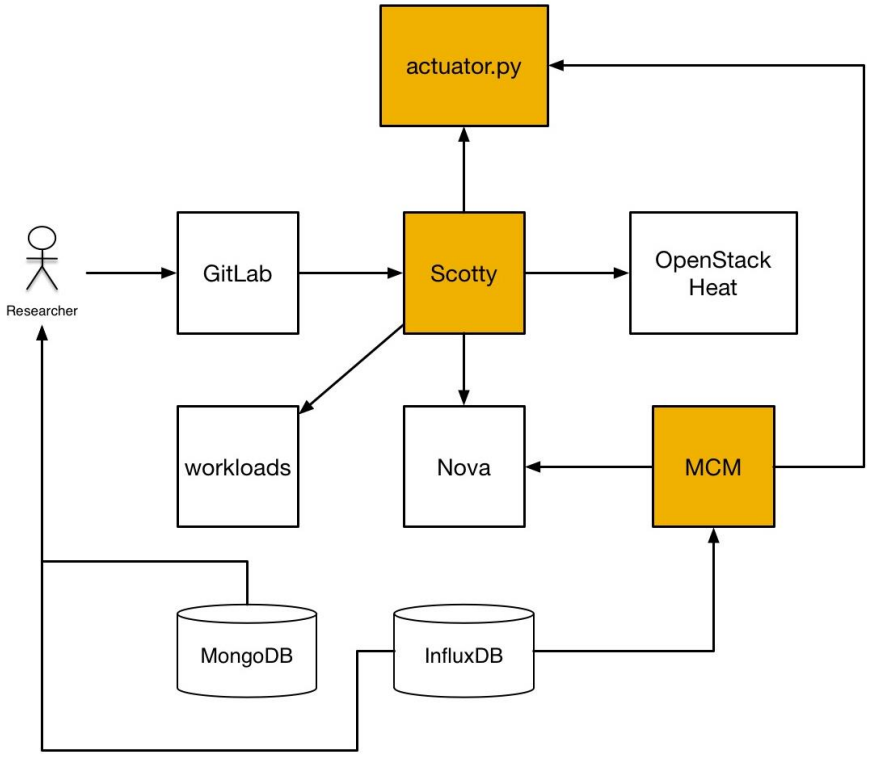


Figure 17: Scotty's integration architecture.

Scotty integrates with a set of third-party components, such as GitLab[21], OpenStack Nova[22], OpenStack Heat[23], InfluxDB[24], and MongoDB[25]. Furthermore, Scotty is integrated with actuator.py and MCM to provide a full experimentation framework for infrastructure research.

The workflow of running an experiment from a component perspective is shown in Figure 18. A researcher performs an experiment by pushing an experiment definition into a repository in GitLab. The repository is located in a specific group, called experiments. The pushed experiment then triggers GitLab CI[26], which in turn triggers a pre-configured GitLab

Runner[27]. The GitLab Runner then executes Scotty with the experiment definition. Scotty uses OpenStack Heat and OpenStack Nova to allocate resources. These resources usually are virtual machines, a private network, and storage. Once the resources are deployed Scotty sets the system state for experimentation using actuator.py. In addition, Scotty sets the MCM strategy to use via actuator.py. During the experiment data is recorded in InfluxDb and MongoDb. After the experiment the researcher can extract the data with scripts provided in conjunction with Scotty. These scripts allow the extraction of quality of service metrics from InfluxDB. Furthermore, these scripts automatically produce box-plots for repeated experiments and descriptive statistics. Furthermore, when applied to different multiple runs of different types of experiments, the scripts can produce an automated inferential statistical analysis, using a t-test. Thus, the researchers get very quick feedback regarding any effect based on the parameters between the two experiment definitions.

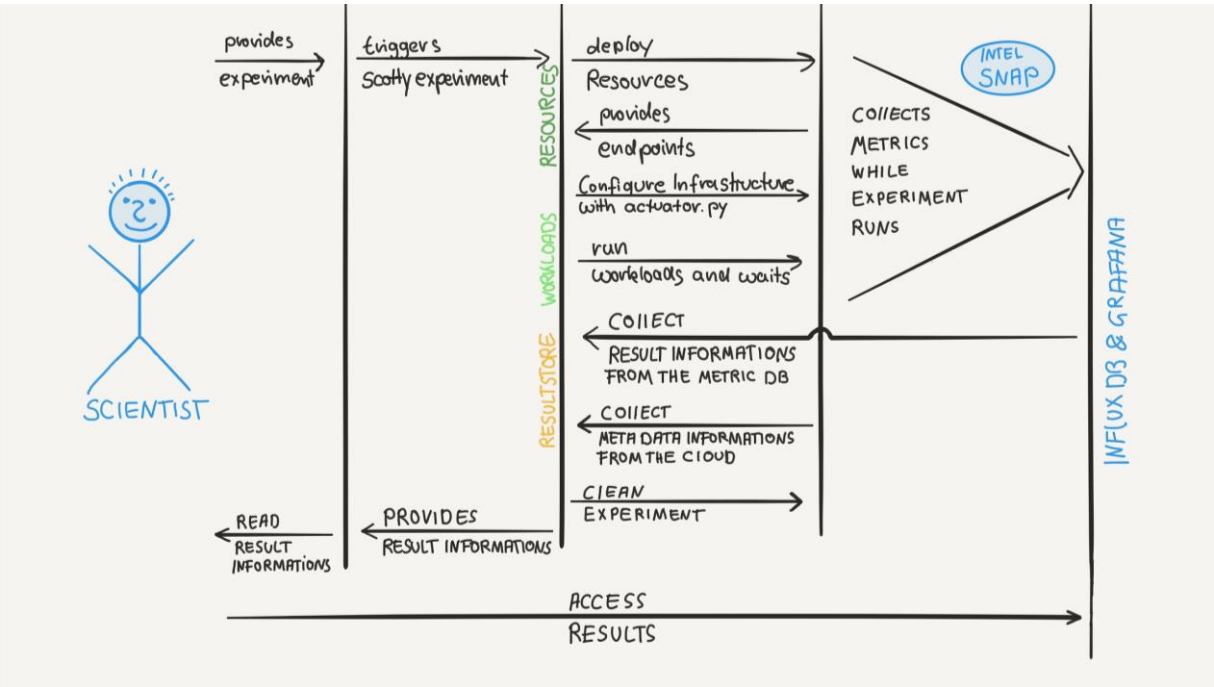


Figure 18: Sequence diagram with Scotty's interactions in the integrated scenario.

The lifecycle of an experiment, of which the above experiment execution is just one phase, is shown in Figure 19. The cycle covers the phases mentioned above. The main intellectual work by the researcher is performed in the data analysis and the experiment definition. These phases are highlighted in the life cycle diagram below. In a typical case the researcher will spend more than 90% of the time in those two phases, where the researcher's knowledge and creativity is required. Without Scotty the researcher will typically spend 70-90% of the time in the blue phases.. The researcher then takes over again with data extraction and with the analysis of the experimental data. Finally, based on the results of the analysis the researcher re-defines the experiment and runs it again, triggering another experimental cycle.

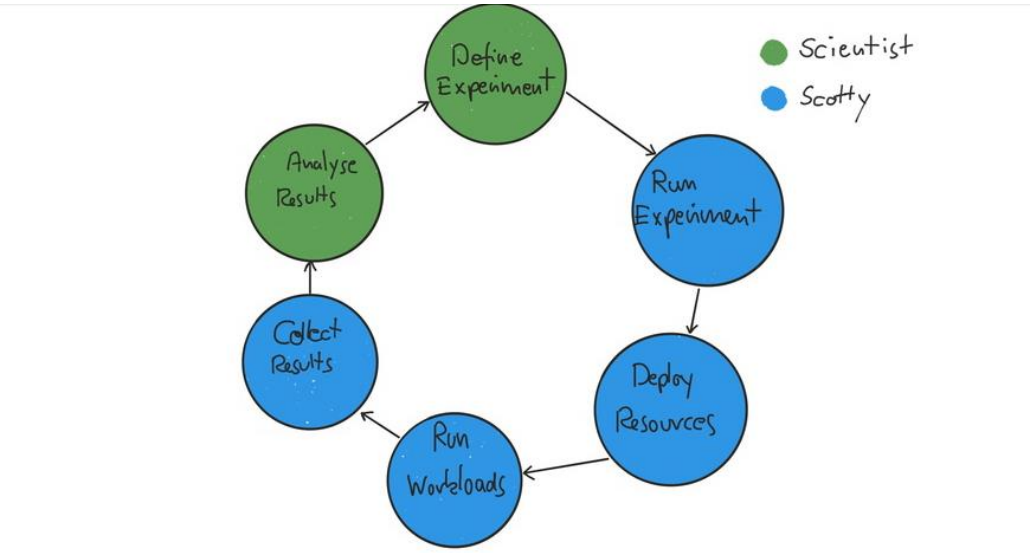


Figure 19: Life cycle of an experiment in Scotty.

Figure 20 provides a more detailed view on the internals of Scotty. Scotty uses a so-called component as central abstraction. Components are abstract. Concrete instances of components are experiments, workloads, resources, and result stores. Those components share basic interfaces, which allows them to be orchestrated in a workflow. The workflow has been described already the previous paragraphs. The resources components define basic resources that are required by workloads to be executed. Examples of resources are a docker swarm installation, a VM with iperf, and a MongoDB instance in a VM. Workloads use resources to execute a workload. Workloads do not perform any significant computation, but trigger computation in the deployed resources. Both resources and workloads are implemented in python modules. These are implemented by third parties and are loaded dynamically in Scotty. Experiments consist of python modules as well. However, in contrast to workloads and resources the python code for experiments is not touched by the user. The standard implementation only reads the experiment.yaml and triggers the resources and the workloads. Thus, the Scotty's users only interact with the experiment definition in the form of the experiment.yaml, python modules for workloads, and python modules for resources. Finally, the result store collects data from the experiment runs and pushes them to a storage backend. Currently S3 and owncloud are available as storage backends.

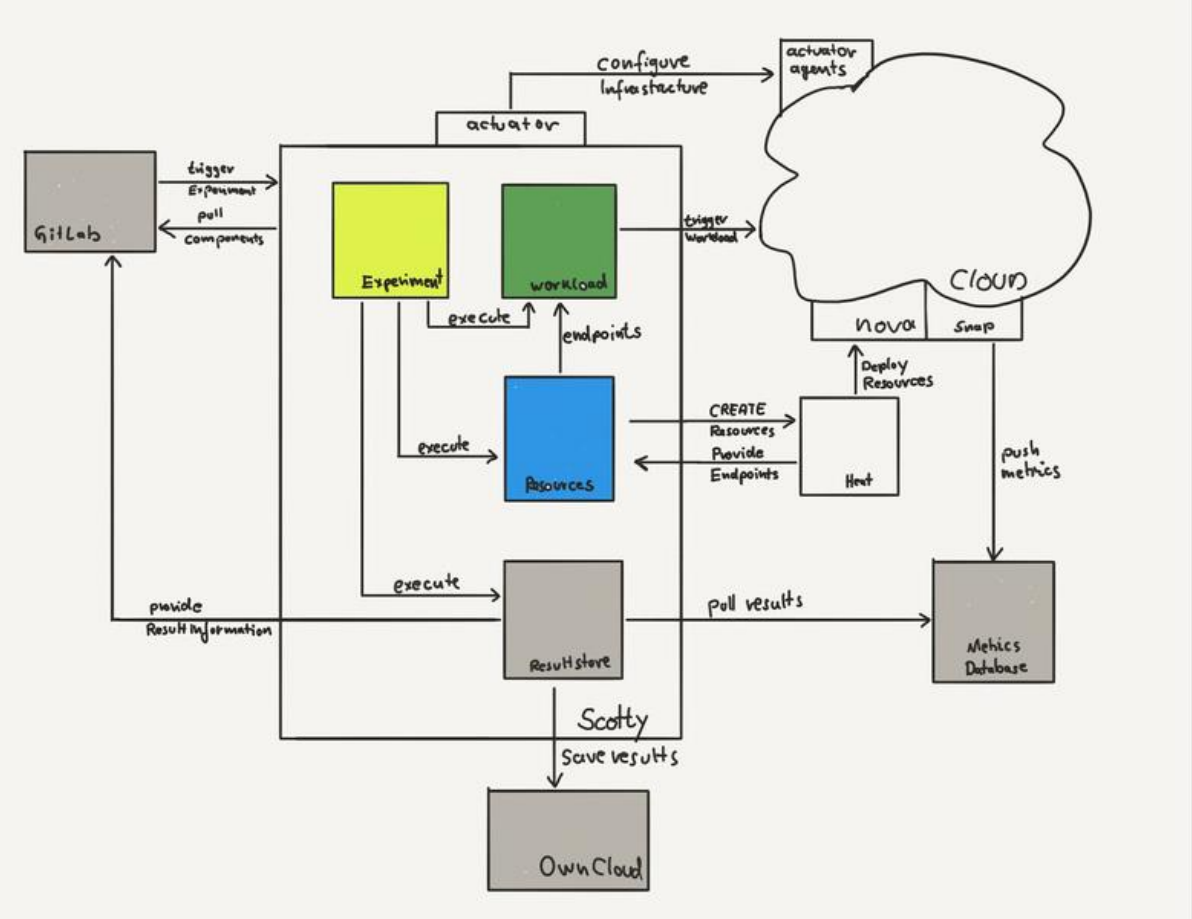


Figure 20: Scotty's internal architecture.

3.3.4 Integrations

Being at the center of various components, Scotty features a large number of components. The integrations required include actuator, IOcm, ioscheduler, taskset, MCM, OpenStack Nova, OpenStack Heat, MongoDB, Snap, and InfluxDb.

Actuator is integrated with Scotty as a control backend. Scotty uses actuator to set the system state before executing workloads. Several components are integrated via actuator. These include IOcm, SCAM, ioscheduler, taskset and MCM. OpenStack Nova and OpenStack Heat are integrated via environment variables and permissive security rules that allow access to those APIs in resources. Finally, Scotty integrates with MongoDB and InfluxDB to store data and extract data for analysis.

3.3.5 Workloads

Workloads can be thought of as a type of apps for the platform. Several workloads have been developed in MIKELANGELO for Scotty. These workloads have partially been developed in

use cases and partially by this work package. The OpenFOAM use case has a Scotty workload to run aerodynamic maps in the cloud. The cancellous bones use cases runs a workload in Scotty that uses HLRS's HPC cluster as a resource.

The data analytics workload has six workloads. Two of those workloads are based on real-world workloads. The other four workloads are based on synthetic benchmarking suites for data analytics. Detailed information about the use case workloads can be found in D6.4.

In this work package, we have developed two workloads, called iperf and CPU stressor. Iperf uses the iperf tool to run IO-heavy workloads whilst the CPU stressor runs a workload on one VM that stresses the CPUs.

3.3.6 Future Work

Scotty's development will be pursued beyond the project. Future work will include the extension of the Scotty workflow, further infrastructure integrations, new user interfaces, and production-grade deployment.

Extending the workflow will include an extension of the analytics pipeline. Currently, the pipeline allows data extraction and simple inferential analysis. Future integrations will include kubernetes for resource provisioning and Telegraf for monitoring. To improve the usability of Scotty, new user interfaces will be developed. Two options for new UIs are a web UI and a prompt-based UI. Finally, future work will focus on the deployment of Scotty as a production service. This will include persisting experiments and storing experiment data for the long term. Experiments then can be identified with persistent identifiers which will allow the use of references to Scotty experiments in scientific publications.

3.4 Actuator.py

Actuator.py is a framework to control hosts and VMs in a cloud infrastructure. Actuator.py is used as a central component by Scotty and MCM. To the best of our knowledge, actuator.py is a completely new concept that can be described as a type of "reverse monitoring". In this section we first describe the motivation behind actuator.py. Then we present actuator.py's distributed architecture. In the following subsection we describe the currently available control plugins for actuator.py. Then we describe the actuator.py's integration with cloud middleware and MIKELANGELO components. Finally, we provide an outlook for future development of actuator.py beyond the project's end.

3.4.1 Motivation

Scotty and MCM need to be able to manipulate various parts of the infrastructure. Scotty needs to set the configuration of hosts and VMs at the beginning of experiments. MCM

needs to configure subsystems on hosts and in VMs. Rather than implement bespoke logic for both Scotty and MCM for every component they need to manipulate, the decision was made to create a flexible actuation framework that could be invoked by MCM and Scotty and indeed other systems, and that could be easily extended to allow them manipulate new components. We call this framework Actuator.py.

As an example, Scotty can take config parameters for experiments that set IOcm and SCAM, among others, into a desired state for an experiment. This state is defined in the experiment.yaml, read by Scotty, and executed by Actuator. In another example MCM can run a strategy that turns on IOcm for certain hosts on demand. Actuator allows IOcm to go beyond the traditional approach of VM placement and makes it a resource manager, by giving it more degrees of freedom to optimize given metrics. To the best of our knowledge, there is no other system like actuator available.

The number of subsystems to control via actuator is very large and hard to enumerate. Furthermore, new subsystems, like IOcm and SCAM, arise regularly. Other subsystems and their interfaces may be proprietary and not publicly available. Thus, actuator needs to be able to cope with unclear requirements for plugins.

Actuator needs to be scalable, because it runs on all hosts in an infrastructure and in most VMs and containers. The number of actuator instances may grow to be a large number. The large number of potential installations also requires failure tolerance, because large numbers of installations are prone to have regular failures.

The next subsection describes how actuator's architecture deals with the requirements for extensibility, scalability, and fault tolerance.

3.4.2 Architecture

Actuator's architecture aims to provide extensibility, scalability, and fault tolerance. The main two abstractions that allow to fulfill these features are a message bus and a plugin system.

To facilitate extensibility, actuator uses AMQP to decouple components, and a plugin system for subsystems. AMQP allows to extend messages seamlessly. Of course, message handlers need to be updated in the slave agents. However, these updates can be performed easily using an auto-update feature, implemented in the slave agents. The auto-update feature is described below in the context of fault tolerance. Extensibility is further provided by a plugin system. The plugin system allows the integration of drivers to control subsystems. As an example, there are plugins for IOcm, SCAM, taskset, and to change the ioscheduler of a block device in Linux. The plugins are loaded dynamically. Thus, one can even load a new plugin from an in-memory file during runtime.

To facilitate scalability, actuator builds on a message bus leveraging the AMQP protocol. The message bus allows for scalability in terms of connected nodes and for a large message throughput. On the client side AMQP offers the benefit of sending asynchronous messages, which opens up a degree of parallelism. Finally, Actuator's reference installation uses RabbitMQ as message broker. RabbitMQ is a common choice for enterprise architectures, such as OpenStack. RabbitMQ offers the benefit of being robust after nearly two decades of use in large production systems. Furthermore, RabbitMQ can be scaled from a single-node installation to a cluster with relative ease.

To facilitate fault tolerance, actuator uses service discovery with AMQP, an auto-update feature, systemd. Service discovery via AMQP allows a master node to discover slave nodes without any central service directory. The overhead to discover nodes is low in comparison to the normal use of the message broker. The slave agents have an auto-update feature. A master node can send an auto-update command to the slave agent. The slave agent pulls the latest stable version of its code from a git repository, restarts its own process, and then signals that everything went well. Considering the potentially large amount of slave agents, the auto-update feature allows to seamlessly update a large number of components. This feature is vital in case any significant issues arise in the agent code. Finally, actuator runs as a systemd service on all nodes. The service is started on boot and is restarted in case of failures. The combination of service discovery, autoupdate, and systemd integration allows actuator to run without major interruptions.

Actuator's high level architecture is shown in Figure 21. The main components in actuator are the master agents and the slave agents. There are few master agents and potentially many slave agents. The master agent is used in an integration with another component, such as Scotty or MCM. In future, we in GWDG may implement a CLI-based prompt UI or a web UI to use actuator interactively. The slave agents are installed on all physical hosts in the infrastructure and in most or all VMs and containers. The AMQP infrastructure forms another important part of actuator's architecture. In our reference implementation, we use RabbitMQ as an AMQP broker. Masters and slave communicate solely via AMQP. In the figure agents are depicted by blue rectangles, exchanges by red circles, and queues by gray rectangles.

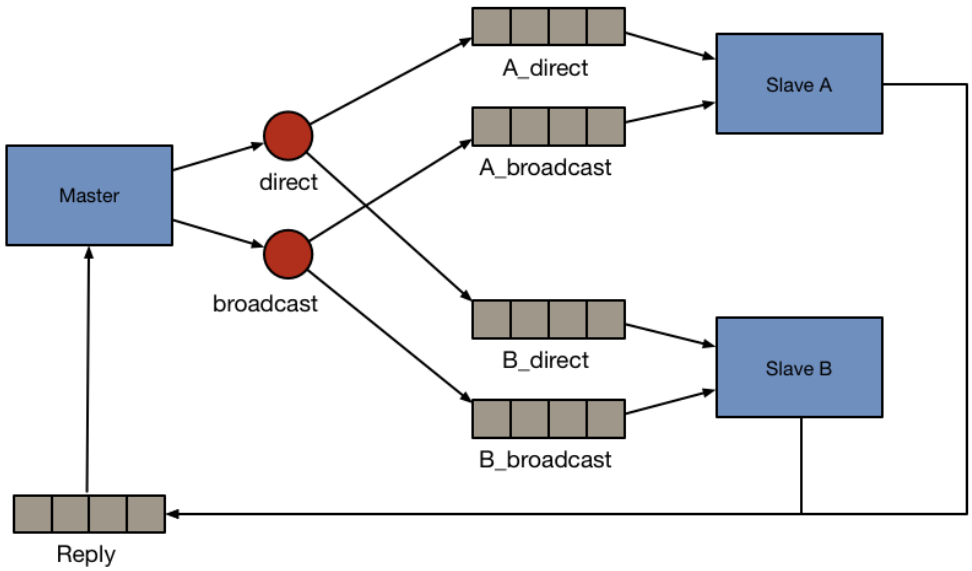


Figure 21: Actuator.py's AMQP-based distributed architecture.

The AMQP infrastructure consists of two exchanges and multiple exclusive queues per node. The infrastructure reflects our two messaging patterns, direct communication and broadcasting. For each messaging mode there is one exchange and one queue per agent. Thus, globally, there is an exchange for direct messages and an exchange for broadcasting. Each of the agents creates two queues of its own. The first queue is for direct messages and the second queue is for broadcast messages. The queues connect to the analogous exchanges. Finally, each master agent has an exclusive reply queue, which connects to the default exchange. The reply queue is used in direct messages. All of our direct messages are performed in RPC fashion. That is, for each direct message from a master to a slave there is a response from the slave to the master's reply queue.

Currently, actuator implements the minimal set of messages, shown in Table 2. The messages allow full control of the infrastructure with a minimal implementation effort. The current set of messages will likely prove to be sub-optimal, albeit sufficient, for many scenarios. However, the choice of messages allows us to evaluate which messages need to be optimized and extended. Furthermore, the minimal set of messages allows us to modify the messaging system quickly. We expect the messaging system to be reworked and settle in a stable setup with time. These types of changes, however will be transparent to users of actuator, since the exposed API in the master agent will hide those changes. The users should only experience performance gains. Furthermore, message changes in future will likely provide more convenient APIs to users.

Table 2: Messages implemented by Actuator

Direct (RPC-style) messages	***Broadcast***
List Plugins	Discover Agents
Get Plugin State	
Activate Plugin Configuration	
Autoupdate	

The slave agents themselves consist of three main parts, as shown in Figure 22: the AMQP integration, a message handler, and the plugin system. The AMQP integration connects to the agent-exclusive direct queue and broadcast with one channel each. The message handler parses and dispatches messages to plugins or an internal handling mechanism. The plugin messages are handled by the plugins and then the response is sent back to the initiating master agent via its reply queue. In case of messages that are not directed to plugins, the slave handles them directly. Currently there are two message types that are handled directly by the slave: autoupdate and discovery. Finally, slave agents are configured in a YAML-based config file. The config file lists the plugins that need to be loaded with configuration parameters for each.

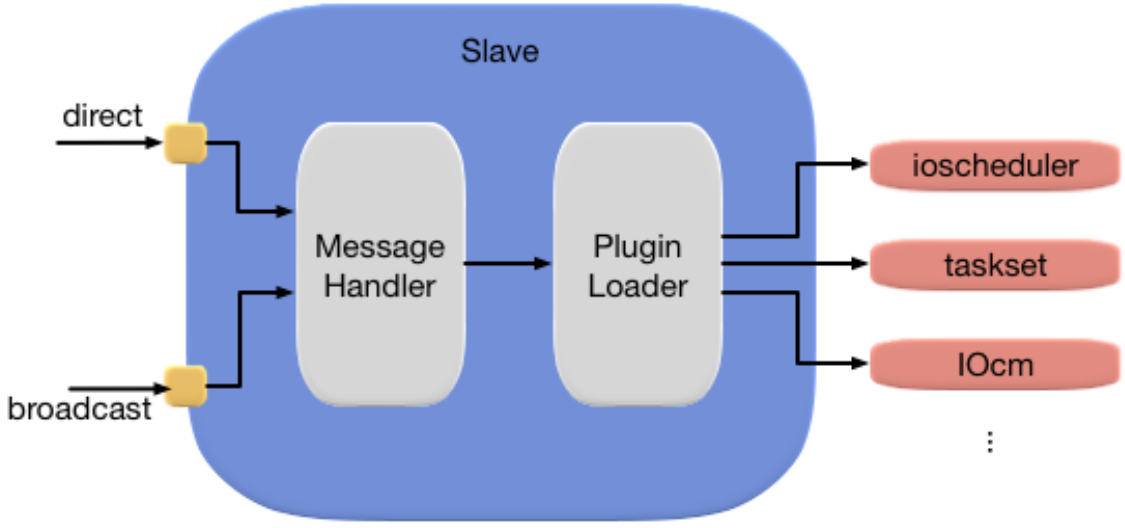


Figure 22: The architecture of an actuator.py slave agent.

3.4.3 Plugins

The currently implemented actuator plugins are ioscheduler, taskset, IOcm, SCAM, and MCM. This set of plugins and its prioritisation is governed by the needs of the project. However, GWDG, the partner responsible for actuator, intend to implement many more plugins over time. The list of plugins that could be used for research already is long. In the next paragraphs we introduce each plugin briefly.

The ioscheduler plugin interacts with the /sys/block interface to query or set the ioscheduler for a given block device. The plugin expects a device in its message. Depending on the type of message the plugin then either reads or activates the ioscheduler for the given block device. Typically available io schedulers in Linux are NOOP, CFP, and DEADLINE. The use of the io scheduler settings can influence throughput and latencies for io operations in workloads.

The taskset plugin interacts with the Linux application taskset[28]. The plugin allows to pin processes to a set of cores or query the current affinity for a set of processes. The plugin expects a string to identify the processes for which the affinity should be set. The string will be matched with the names of all active processes. Taskset should be used in VMs and containers. On physical hosts pinning should be done for VMs and needs to be managed by libvirt. Process pinning via taskset can reduce interference between processes in the CPU cores and in the L2 cache.

The IOcm plugin allows to turn IOcm on the physical hosts on or off. Due to the limitations of the IOcm control scripts the communication is only one way. Thus the IOcm state can be set, but it cannot be queried.

The SCAM plugin allows to turn SCAM on and off on the physical hosts. The plugin can query and set the state of SCAM's monitoring and noisification mechanisms. Thus, actuator can be used to first monitor for attacks on separate hosts with low overhead. Then, in case suspicious activity is detected, actuator can be used to directly start the noisification on the targeted hosts.

The MCM plugin allows actuator to load strategies for MCM. The plugin copies an MCM strategy into MCM's path, activates the strategy, and finally restarts MCM. The MCM plugin is used to facilitate research of resource management strategies via Scotty. Scotty uses the MCM plugin to load the MCM strategy specified in the experiment.yaml.

The stressor plugin uses the stress-ng tool[29] to stress designated subsystems in the host. The plugin takes configuration parameters analogous to those defined by stress-ng and forwards them to the stress-ng binary. Thus, the plugin allows the user to stress an arbitrary

host subsystem, such as L2 cache, instruction pipeline, memory, and NIC with a high degree of fidelity. Stress generation as provided by stress-ng is an important tool to assert the sensitivity of workloads to stresses. Furthermore, stress-ng can be used to simulate resource interference in a workload.

3.4.4 Integrations

Actuator integrates with other components in two ways. In the backend actuator integrates with subsystems in hosts, VMs, and containers via the above-mentioned plugins. Actuator itself, however is integrated in other components. By design actuator is most useful as a utility to be used in other software. In MIKELANGELO, we use actuator in Scotty and in MCM.

Scotty uses actuator to set up infrastructure state before experiments. The configuration for actuator is defined in the experiment.yaml. Then, in Scotty actuator is used in a special phase of the experiment life cycle. First, all agents are discovered with a broadcast call. Then, the configuration on each agent is set as defined by the experiment's configuration. Currently, Scotty defines only global settings, that is each plugin settings is activated on all discovered slave agents that have loaded that plugin.

MCM integrates with actuator in a special actuator wrapper. MCM does not use Scotty by itself, but its strategies do. Researchers can use the actuator wrapper to activate configurations for plugins in individual slave agents.

3.4.5 Future Work

In the future, GWDG plan to extend actuator with more plugins, improve the messaging API for more convenience, and investigate the integration with Apache Zookeeper[30] to provide another degree of fault tolerance in large installations such as our production cluster. The most important future effort for actuator will be the extension of its plugin catalogue. With each plugin the benefits of actuator will grow. Furthermore, once more data on the use of the actuator API will be collected, the messaging API will be extended, which is expected to be minor work. Finally, as the actuator installation will grow in size, GWDG will investigate an integration with Apache Zookeeper, to provide another degree of fault tolerance in our cluster.

3.5 MCM: A live resource manager for the cloud

MCM is a standalone service which enables live-scheduling of VMs in OpenStack. Live-scheduling in this context means the automated monitoring and subsequent live-migration and resizing of VMs to reach certain objectives. Objectives include examples such as load balancing and server consolidation. We are integrating MCM into the GWDG OpenStack

Testbed. We plan to offer a programming environment in which researchers have quick and easy access to metrics and actions from both OpenStack and Snap. We aim to enable researchers to quickly iterate and evaluate new scheduling algorithms.

3.5.1 Motivation

MCM provides three main advantages for MIKELANGELO and cloud research beyond MIKELANGELO: capabilities for live resource management, interoperability, and integration with Scotty.

Live resource management poses two requirements. First, operations have to happen online with short latencies. The system needs to be able to react to events in the infrastructure. Second, operations for resource management should go well beyond the traditional VM placement strategies.

Interoperability poses two requirements as well. First, MCM should interoperate with different infrastructure backends. The current implementation focuses on OpenStack. However, in future an integration with kubernetes and other platforms would be of interest. Second, MCM should be interoperable with different metering systems. In the project, MCM focuses on an integration with InfluxDB, with data in Snap's format. In future, however, other time-series databases or other monitoring solutions might become relevant. Thus, MCM needs to be able to adapt to those changes quickly.

An integration with Scotty would allow MCM to be used in Scotty experiments. In such a use case a researcher would define an MCM strategy in the experiment repo. Scotty would then automatically load the strategy while performing the experiment. This workflow would allow researchers to evaluate experiments very quickly.

3.5.2 *Architecture*

Like OpenStack, MCM is written in Python. It is designed to work with Snap, Nova, and actuator.py. Although, due to the modular approach, support for other components can be added later on. Nova is the component of OpenStack responsible for compute services.

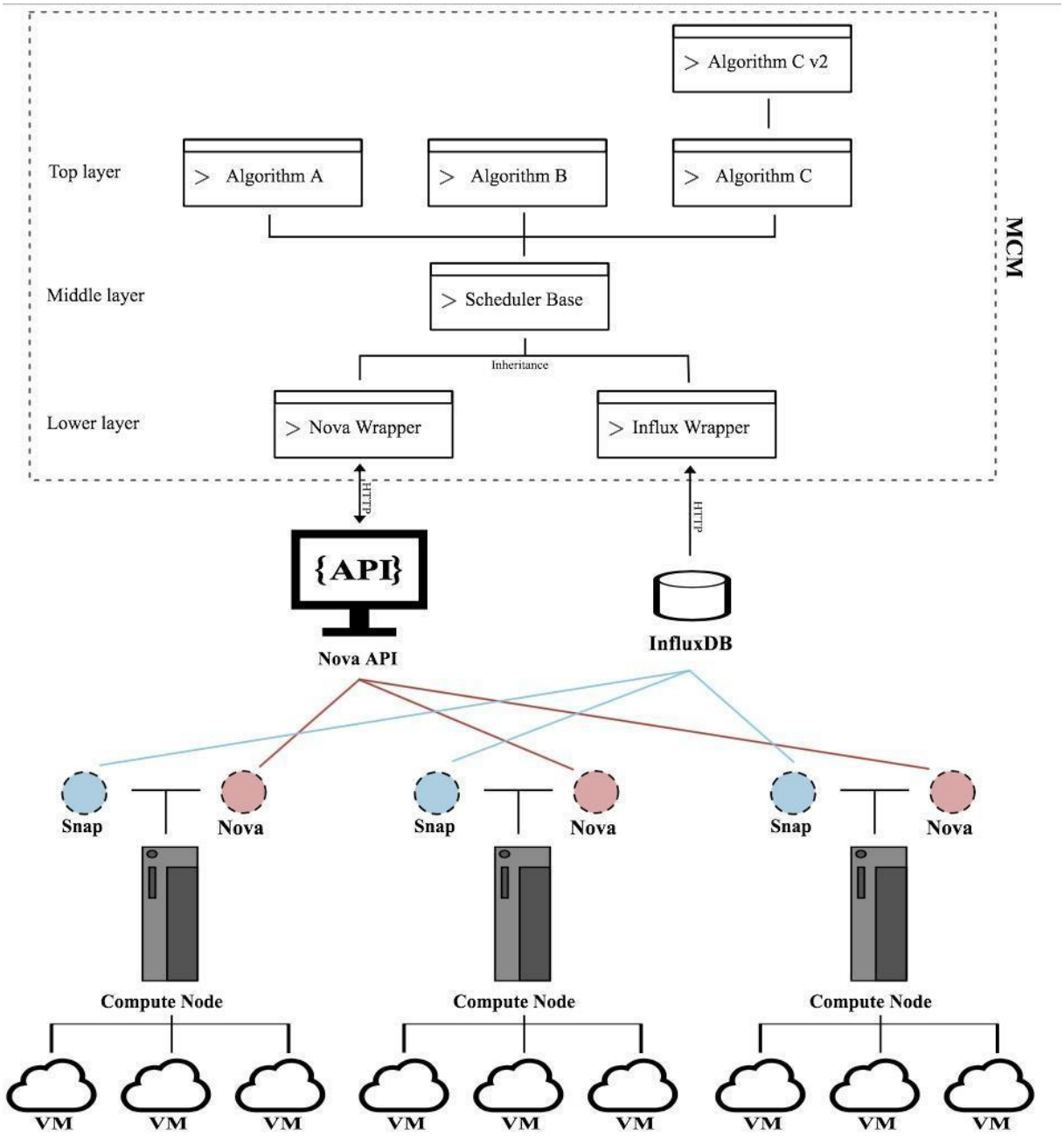


Figure 23: MCM's system and integration architecture.

On the lowest layer MCM wraps around the Python programming-APIs for InfluxDB and Nova, which, in turn, wrap around the respective HTTP API. It simplifies, abstracts and partially enhances those APIs. This is done to accelerate development of new algorithms: Researchers get a flat, unified and easy-to-learn interface for both cluster control and metric retrieval. This is achieved through self-explanatory function and argument names, and in-code documentation. It also decouples the actual scheduling code from the metric-and-control-code, making the former more portable. Additionally, this rather loose coupling

results in an easy deployment, as only a Python interpreter and network access to InfluxDB, Nova, and actuator.py is necessary to run MCM. There is also some work done on basic code generation for highly structured code, like for example metric retrieval from Snap: Snap supports easily adding and removing metrics from being collected. A code generation component could accommodate this by dynamically adding and removing parts of the MCM API without adding much complexity.

On the middle layer this information is synthesized. All functions which collect, interpret or simulate aspects about the cluster state live here, as long as they are not tightly coupled to a certain scheduling-algorithm. Initially this will include functions to get comprehensive metric aggregations and basic estimations about how actions will influence the cluster state. Later on this layer could also include more sophisticated components like machine-learning-based forecasting, cluster state scoring, or network topology awareness.

On the top layer are the implementations of the actual scheduling algorithms. These are also written in Python and just need to implement a schedule() function to qualify as a runnable Scheduler for MCM. They have access to all the functions exposed by the lower and middle layer. Currently the scheduler modules are dynamically loaded from the MCM code-base by Python using a simple config option. Beyond MIKELANGELO this could be enhanced by adding other interfaces to control MCM or even add code on-the-fly.

This architecture is realized via native Python inheritance instead of a plugin-based structure. The latter has slightly less repetition and is cleaner when it comes to code-reuse and – at least in theory – abstracts more of the inner working of the software. However, the inheritance-approach is more flexible, simpler to enhance and a better fit for changing environments and requirements. Additionally, inheritance-based code inspection and completion is usually supported out-of-the-box in Python IDEs which also makes development and usage easier and quicker.

3.5.3 Integration

Although MCM's core architecture is set on loose coupling, it integrates well with many components from MIKELANGELO and the broader ecosphere. From the viewpoint of MIKELANGELO MCM integrates with Snap, actuator, IOcm, and SCAM. Beyond, MIKELANGELO MCM integrates with OpenStack, InfluxDB, and actuator's plugins.

The integration with Snap manifests itself in the metrics that MCM uses in its strategies. The data is collected using Snap and it is then written to InfluxDB. MCM regularly reads the data from InfluxDB to make resource management decisions. Although MCM does not directly interact with Snap, it is dependent on the data collected by Snap.

The integration with actuator is bidirectional. First, MCM has an actuator wrapper. The actuator wrapper allows MCM strategies to control the infrastructure via actuator. Second, MCM integrates with actuator via an actuator plugin for MCM. This plugin allows actuator to load a given MCM strategy via actuator.

IOcm and SCAM are integrated in a very similar way with MCM. Both are not used directly by MCM, but via actuator as a proxy. Actuator provides plugins for IOcm and SCAM, which can be used to turn specific features in IOcm and SCAM on or off.

The integration beyond MIKELANGELO components include integration with OpenStack, InfluxDB, and actuator plugins. The integration with OpenStack happens via the nova wrapper. The OpenStack integration allows MCM strategies to migrate and scale VMs. The integration with InfluxDB allows strategies to query data from InfluxDB, independent of the data source. Finally, the integration of actuator plugins allows for control of specific subsystems. The current set of plugins used by MCM includes ioscheduler, stressor, and taskset. These plugins are described in Section 3.4.3.

3.5.4 Future Work

Beyond the MIKELANGELO project lifetime, future work on MCM will revolve around interoperability, development of strategies, and potentially extensions for production use.

For the short term perspective, interoperability for MCM will likely deal with integrations with kubernetes and Telegraf. A kubernetes integration would allow the evaluation and development of resource management strategies in kubernetes. The use of kubernetes is interesting as kubernetes and container-based infrastructures are becoming an established technology for certain areas of infrastructure management. An integration with the Telegraf metering system is interesting because it has a large user base. The changes between Snap and Telegraf in terms of recorded data are not very big, thus this migration should happen relatively easily. These two integrations would probably allow an easier adoption of MCM in many infrastructures.

The development of strategies for resource management will be the main focus on MCM. MCM and its vital integrations, actuator and Scotty, have reached a sufficient maturity to perform serious experimentation. After the project ends GWDG intents to continue to use this combination of components for research in resource management.

Finally, when good strategies for resource management can be found, GWDG will prepare MCM for production use. The extensions necessary for that would revolve around extended testing, CI integration, and integration with the deployment process of the targeted platform, such as OpenStack, kubernetes, and others.

4 Cloud Integration

The cloud integration in MIKELANGELO spans hardware, generic software, and MIKELANGELO components. Since MIKELANGELO only deals with software, the hardware components are off-the-shelf servers. The generic software layer deals with standard cloud installations, which touches the cloud middleware, software-defined networking, and the hosts' operating systems. Finally, the MIKELANGELO components need to be mentioned separately, because the integration revolves around them. The MIKELANGELO components can be found in the hosts, in the cloud layer, and even in the application layer. The integrated MIKELANGELO components are IOcm, SCAM, Snap, OSv, LEET, Scotty, actuator, and MCM.

To enable the cloud integration various new components have been developed in work package 5. These new components are Snap, LEET, MCM, actuator, and Scotty. Scotty is a special case since it has been developed in conjunction with work package 6. Details on the individual components are described in Section 3.

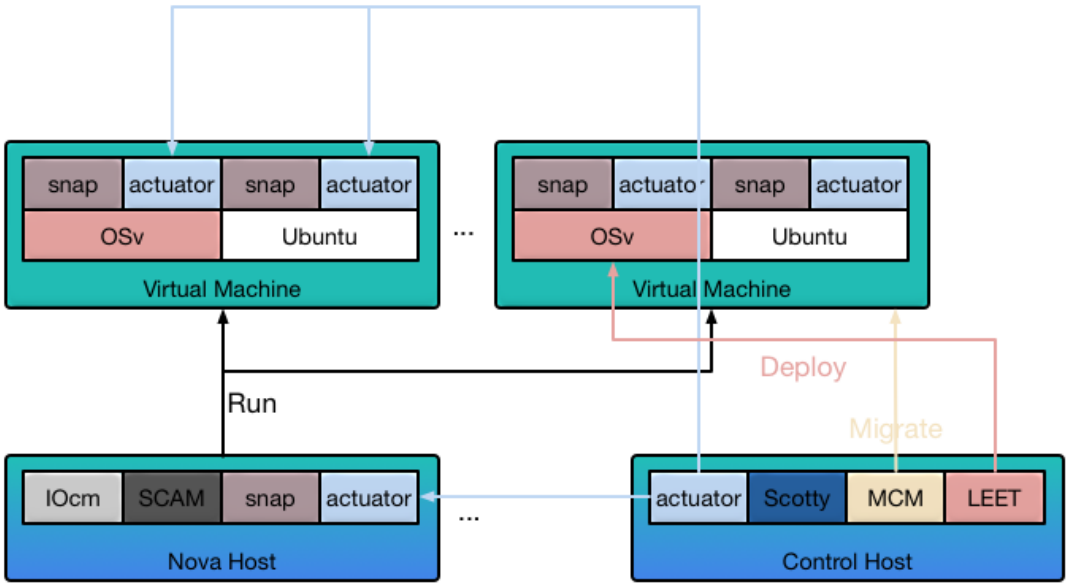


Figure 24: The cloud integration architecture with all integrated MIKELANGELO components.

4.1 Hardware

The hardware layer consists of six identical servers. The servers run an Intel Xeon CPU (E5-2630) at 2.4GHz. The CPU has eight cores and supports 16 hardware threads. The L2 cache is 256KB large and the L3 cache is 2MB large. Each node contains 128GB of memory, which allows running of a reasonably large deployment of VMs. Finally, each server has a dual-channel 10GBit NIC. The NICs are connected in dual-channel mode to a sufficiently sized

switch. The switch is a Cisco Nexus switch that routes production data throughout the network in conjunction to the testbed. However, the testbed's traffic is isolated from the production traffic.

4.2 Software

The software layer consists of the operating systems on the hosts, the cloud middleware, extensions to OpenStack, and some supporting components.

The hosts run Ubuntu 14.04 as their operating system. Ubuntu 14.04 is a long-term support version from Ubuntu. The main benefits of using an LTS version are the prolonged security patches and backports of new packages. When the project started, Ubuntu 14.04 was agreed to be used as host operating system for the test beds. In the meantime Ubuntu 16.04 LTS came out, which would have been a viable alternative. However, early on the consortium also agreed to use a 3.18 Linux kernel as a basis. This dependency manifests itself in IOcm. This dependency also constrains the choice of host operating system. Running Ubuntu 16.04 with our custom kernel would be hard to do, if at all feasible. Pinning the version of Ubuntu to 14.04, in turn, affects the version of OpenStack.

The cloud middleware installed in the test bed is OpenStack Liberty. OpenStack Liberty was the newest version of OpenStack, when the testbed was deployed. However in the meantime new versions of OpenStack have been released. These releases include a large number of new features. Unfortunately. an installation of those new releases was not feasible, because the testbed runs Ubuntu 14.04 on the hosts.

In addition to the vanilla OpenStack installation, the cloud middleware includes OpenStack Heat and OpenStack Sahara. Heat has been installed for cluster deployment and management. Heat is used often by Scotty resources. For example, the iperf-workload uses Heat templates to deploy client and server VMs, and to install and configure software in them. Heat is also used by LEET and by Sahara. Sahara is an abstraction on top of Heat, which provides facilities to deploy a big data cluster.

Finally, the testbed features scalable installations of MongoDB, InfluxDB, MCM, and Scotty. MongoDB is used by Scotty to store static monitoring data at the beginning of each experiment. InfluxDB is used as time series database for Snap. MCM runs as part of the cloud middleware to allow algorithm research for research management. Scotty runs as part of the cloud middleware as well and integrates with InfluxDB, MCM, and OpenStack.

4.3 Integrated MIKELANGELO Components

The cloud testbed integrates with MIKELANGELO components throughout all software layers. The integration includes IOcm, SCAM, Snap, actuator.py, OSv, LEET, and Scotty. IOcm runs in

the hosts' kernels with extensions in the form of userland scripts. SCAM runs on the hosts in userland and in VMs with special images, as part of the cloud deployment. Snap's installation is distributed across the whole installation. Snap runs collectors on the hosts and in the VM. In a special management domain of the cloud installation, Snap runs the processing, and publishing. Actuator is installed on the hosts and in many of the VMs in the testbed. OSv runs in VMs with a custom image. LEET is installed in VMs and uses the OpenStack installation. Scotty runs in the management domain in a container. The next sections briefly summarize the integration of MIKELANGELO components in the cloud testbed.

4.3.1 IOcm

IOcm is deployed directly in the host OS on all hardware nodes. The physical hosts run a custom-built kernel with the IOcm modules. As part of the integration work, we have built the kernel and deployed it on the hosts. Furthermore, we have deployed management scripts for IOcm on all hosts. These scripts allow IOcm to be turned on and off. Actuator uses these scripts to modify the IOcm state. One downside of the current installation of IOcm is that it does not cope well with virtual machine migrations. Thus, we need to prevent VM migrations when IOcm is turned on. Furthermore, there is no reliable way to find out whether IOcm is active and with how many cores. As a consequence, the actuator plugin for IOcm throws an exception when queried for the state.

4.3.2 SCAM

SCAM is deployed directly on all physical hosts in the cloud testbed. SCAM runs in user space, which makes the deployment easy. The monitoring and noisification with SCAM are easy to deploy and work without issues. The side-channel attack with SCAM requires tweaking of parameters when deploying onto a new architecture. Thus, the attack is harder to port than the monitoring and mitigation. Scam further integrates in the cloud with Snap via a collector plugin and with actuator via an actuator plugin. The actuator plugin allows the control of SCAM monitoring and noisification.

4.3.3 Snap

Snap is deployed on all physical hosts, in most VMs, and in the management domain of the cloud testbed. The hosts and VMs run Snap collectors, while the management domain runs a processor and published. Snap is the central data source for time series data in the cloud testbed. The Snap installation publishes its data to the cloud testbed's InfluxDB installation. The data collected through Snap is, in turn, used by MCM and Scotty. MCM uses the data in its strategies. Scotty extracts the data from InfluxDB to allow for the quantitative analysis of experiments.

4.3.4 OSv

OSv's integration is a simple deployment as an OpenStack image. The integration merely requires to upload OSv and to register it as an image. This process has been simplified during the project by an extended support for cloudinit. Furthermore, OSv is used by LEET for application deployment. The work on OSv and LEET have been largely performed as part of work package 4.

4.3.5 LEET

LEET integrates with the cloud testbed via OSv and via the use of the OpenStack APIs. LEET builds on OSv to deploy applications. To provision the required resources in OpenStack, LEET integrates with the OpenStack API. Details on LEET and its value as integration enabler can be found in Section 3.2.

4.3.6 Scotty

Scotty uses the cloud testbed as its main backend to perform experiments. Scotty itself runs in the cloud testbed's management domain as a service. Furthermore, Scotty provides integration APIs to leverage OpenStack's APIs for Nova and Heat, to provide resources for experiments. Scotty's integration with OpenStack allows running a large number of infrastructure experiments quickly and without large effort. These experiments can help to identify bottlenecks in a system and also to boost research into more efficient management of virtual infrastructures.

4.3.7 Actuator

Actuator runs on all physical hosts, in most VMs, and in the cloud testbed's management domain. Slave agents are installed on the physical hosts and in VMs. The master agents are installed in the testbed's management domain as part of Scotty and MCM. Both, Scotty and MCM have their own master agent. The integration with actuator allows fine-grained control over cloud resources in Scotty and MCM.

4.3.8 MCM

MCM runs in the cloud testbed's management domain as its own service. MCM integrates with OpenStack's nova API, with actuator, and with Snap via InfluxDB. The Nova API is used for VM placement and VM migrations. Actuator is used to allow for fine-grained control of plugins on hosts and in VMs. Snap is used to collect metering data, to perform data-based decisions in resource management strategies.

5 HPC Integration

During the first year of MIKELANGELO the focus was on the identification of requirements, and a first proof-of-concept prototype, demonstrated at the first review. That prototype has been demonstrated at the first review in February 2016, and the details of this early stage are described in deliverable D2.19[31].

The second and third year of the project focused on the fulfillment of jointly prioritized requirements. In year two of the project our prototype matured, became more stable and feature rich. Furthermore, the accomplished results on bash-based logging became a separate project, called log4bsh, and were published. Also an initial version of vTorque was published in the second year.

This document covers the implementation related to the final architecture, as described in deliverable D2.21. The final state of components integrated, the testbed used for validation and evaluation, as well as an overview about the continuous integration for HPC are also presented, together with the current state of the testbed and code management infrastructure.

5.1 HPC Testbed Infrastructure

In the last year of the project, the HPC testbed infrastructure has not changed, since it covers all our needs.

Virtualized services run on a physical front-end, these comprise performance monitoring (grafana and InfluxDB), CI-based experiment execution (Scotty on Jenkins host, see section 5.5), and a virtualized node for job submission and monitoring. This setup has the benefit that services can be migrated to spare nodes in case the physical host breaks, restoring access to the testbed in a short time. The shared storage is a ZFS raid residing on the physical host, and is exported via NFS, thus failing disks can be replaced without losing any data.

Failing compute nodes are not a major issue, as they are simply replaced by new hardware, as the rest of the cluster is still available for jobs, and the compute node's installation can be done by cloning from another node's disk.

The Jenkins slave requires access to the code repositories for fetching the hpc-workload-gen and experiment definitions for the use cases. These can reside on the Jenkins host locally, but may also be hosted on an any external server, like github.com.

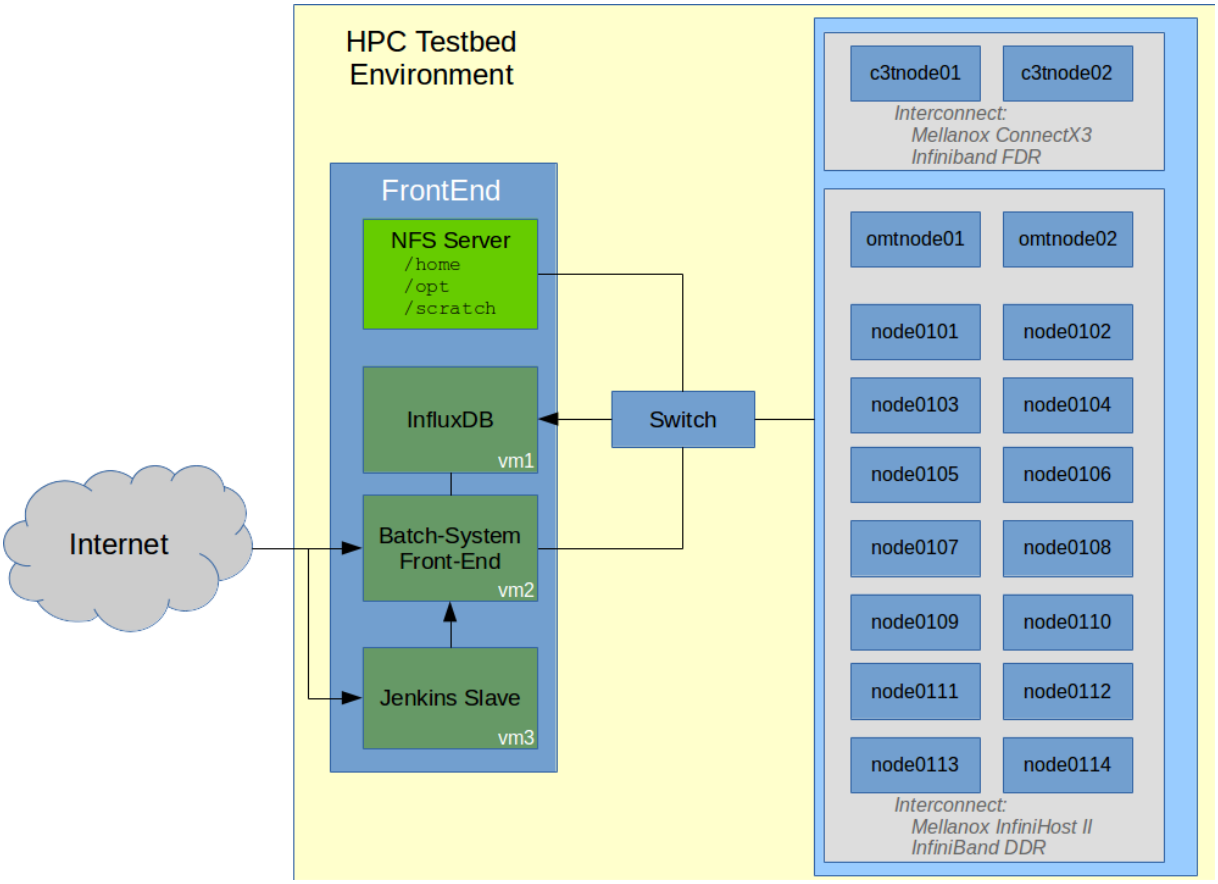


Figure 25: HPC testbed overview

Compute Nodes

There are three different kind of compute nodes, c3t-nodes are equipped with more recent infiniband cards (ConnectX3, FDR) than the other compute nodes, required for vRDMA integration and testing.

The omt-nodes are intended for custom kernel building and testing, and are as all other nodes equipped with older infiniband hardware (InfiniHost II, DDR). The `node0101` and `node0102` are build-system nodes providing all packages required for compilation of source code.

Storage Systems

Shared storage for compute nodes is NFS based, user homes are located on the physical front-end host, in addition there is a dedicated SSD for fast intermediate data, located at a separate storage server. However, NFS is a bottleneck if more than one batch job is running, as they then compete for IO cycles. A scalable highly parallel file-system, i.e. Lustre, was not justifiable from an economical point of view (license costs, procurement costs, maintenance costs), even though it would have been desirable from a developer's perspective.

5.1.1 Requirements for vTorque

PBS Torque[32] is required to be installed for a successful deployment of vTorque. Torque checks for some optional files, like for the root pro/epilogue expected in a specific place, with correct file access rights. These scripts are executed shortly before and after the execution of the user's job script and optional user pro/epilogues. As the files are executed as root, no access by users is desired, and for security reasons they aren't executed when permission does not match. They are used to prepare the infrastructure, in the case of vTorque to prepare the virtual environment and clean up after the run, however the scripts previously in place will also be executed, since they are just wrapped by vTorque scripts.

To simplify the deployment of the vTorque scripts hooks, symlinks are in place. They are linked from the source code in the shared folder to '`/var/spool/torque/mom_priv/`' on each of the compute nodes to the cluster wide vTorque installation on a shared file-system.

vTorque requires the following software and infrastructure to work properly

Table 3: Software prerequisites for vTorque

Package	Requirements
`qemu-kvm`	Required on all compute nodes.
`libvirt-bin`	Required on all compute nodes.
`cloud-utils`	Required on all compute nodes.
`cloud-init`	Required to be installed in VM images, with data source `nocloud` enabled.
`arp`	Required on all compute nodes.
`realpath`	Required on the front-end and all compute nodes.
`bash`	Version >=4.0 required on the front-end and all compute nodes.
`log4bsh`	Required on the front-end and all compute nodes.

Table 4: Infrastructure prerequisites for vTorque

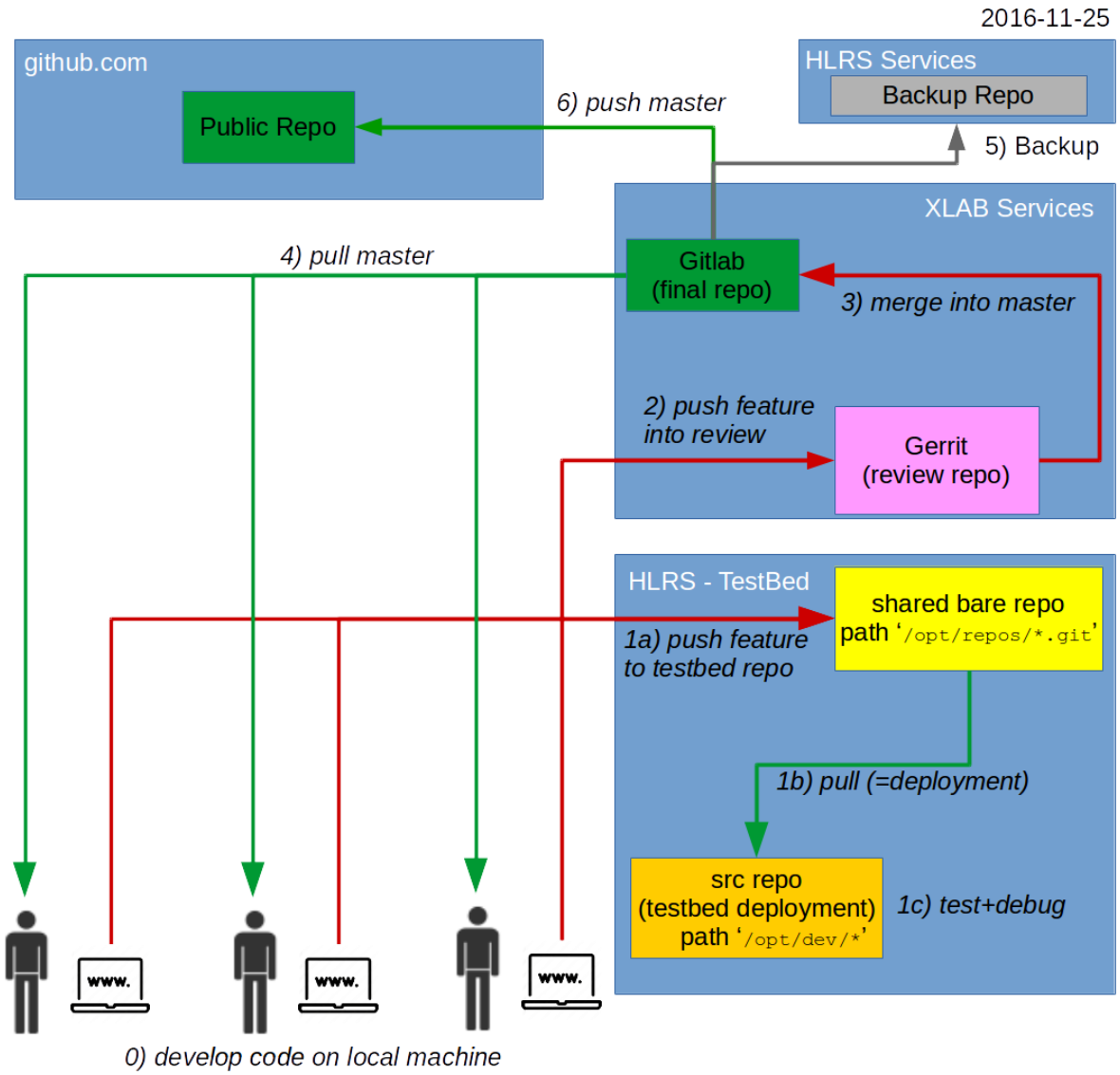
Requirement	Description
`shared /opt`	A shared storage, available on the front-end, head node and

	compute nodes. In our setup the head and front-end node are the same. vTorque is deployed there.
`shared $HOME`	Used for temp files and vTorque log.
`shared workspace`	Fast, shared storage for intermediate data.

5.1.2 Development Workflow

The management of vTorque source code and its deployment on the USTUTT testbed has been verified to meet all our needs. Elements include a Gerrit[33] installation to make a code review simple, GitLab as a main repository, FusionForge[34] as an offshore backup hosted on HLRS' premises and GitHub[35] as a publication platform.

In the improved workflow (see Schema of the git workflow setup Figure 26), the upstream repo is hosted at XLAB and connected to Gerrit. This setup provides the benefits of code reviews and further, has a clear upstream process for the public access to the source code, while the testbed repo is a separate repo dedicated to testing and debugging, which can be thrown away and cloned from scratch without losing any former work.



MIKELANGELO HLRS git Workflow

0) develop code on local machnie

1a) push to FrontEnd (bare repo)
1b) checkout on testbed (= deployment)
1c) test+debug

2) push validated code into review

[If review not passed, repeat steps]

3) merge feature into master branch

4) pull master branch to local machine, create new feature branch, start with 0) again

5) backup to HLRS project's server

6) push code to public repo on github.com

Figure 26: Development workflow

5.2 Final Architecture Implementation

In year one of the project the initial requirements for the HPC integration were captured and a first simple architecture design was implemented as a proof of concept prototype.

During the second year the preliminary architecture evolved as described in deliverable D2.20[36]. The integration of several MIKELANGELO components relevant to HPC started, as well as the continuous integration for HPC use cases, besides updates in our HPC testbed and the code management infrastructure for vTorque.

The implementation of the final architecture, as described in D2.21, covers most of the targeted functionality, with a few nice-to-have features cancelled. A detailed overview about all new developments surrounding vTorque is given in the following sections. The focus of the description is on enhancements and modifications, rather than repeating all content presented in the previous deliverable D5.2.

5.2.1 Requirements for HPC Virtualization

The table below is the full list of all low-level requirements that have been identified for HPC, or to be more exact for vTorque, the virtualization layer for Torque-based batch systems.

Table 5: Requirements for HPC integration

Requirement	**Description**	**Comment about status**
R5.PBS.001	A switch for Infiniband mode for vRDMA	Implemented, can be considered a tech preview, not production ready.
R5.PBS.002	A switch for RoCE mode for vRDMA	vRDMA prototype 3 integration not completed.
R5.PBS.003	Configuring base image for PBS	Successfully tested.
R5.PBS.004	Unified contextualisation via CloudInit	Successfully tested.
~~R5.PBS.005~~	~~A switch to enable SCAM support~~	Relevance of SSL in HPC environments is not crucial.
R5.PBS.006	List all available image files	Successfully tested.
R5.PBS.007	Select image to use for jobs	Successfully tested.
R5.PBS.008	Select jobscript	Successfully tested.
R5.PBS.009	Select virtual resources for execution	Successfully tested.
~~R5.PBS.010~~	~~Select hypervisor~~	The hypervisor is always KVM, sKVM features are configurable

		independently.
R5.PBS.011	per VM resource definitions	Successfully tested.
R5.PBS.012	checkpoint+restart for virtualized jobs	Skipped.
R5.PBS.013	node health monitoring	Intended for triggering live migration, however not integrated, due to live migration not in place.
R5.PBS.014	spare node mgmt	There is no added value without live migration.
R5.PBS.015	job exclusive, virtualized and isolated networking	Skipped, due to required hardware not available.
R5.PBS.016	live migration of virtual guests	Preliminary work.
R5.PBS.017	support for user provided metadata	Skipped, as it is considered a nice to have feature, other requirements are more important.
R5.PBS.018	prevent users from setting any uid and access other users confidential data stored on a NFS	Successfully tested.
R5.PBS.019	prevent users from booting images	Successfully tested.
R5.PBS.020	setup / installer	Successfully tested.
R5.PBS.021	Generic PCI mapper hook script.	Skipped.
R5.PBS.022	Support for UNCLOT (OSv, only)	Integrated.

5.2.2 vsub

Previously there was no `vsub` cli, but qsub was wrapped. As this has an impact on the job-throughput per second, which is reduced by the wrapper due to argument parsing, the decision was made to have in parallel to Torque's `qsub` a dedicated cli for VM jobs, called `vsub`.

This way there is no direct impact on the existing Torque's qsub (for bare metal jobs), since they are no longer routed through a wrapper. Nevertheless a few seconds overhead is added to the root prologue, used for node preparations, which runs shortly before the actual job.

Since flat file flags written onto a shared storage (e.g. NFS) are required in order to detect if it is either a bare metal or VM job, the root prologue may take up to `NFS_TIMEOUT` seconds (a newly introduced configuration parameter, see section 5.4.1) compared to a Torque-only setup.

5.2.3 vmgr

The cli tool vmgr is newly introduced and is named vmgr as a counterpart for the qmgr (Torque's queue manager) and it serves the following purposes:

1. Print vTorque configuration (displays enabled features and defaults)

```
vmgr show config
```

2. Provide admins a tool to manage images for job submission

```
vmgr add <image> [<description>]
vmgr update <image> <description>
vmgr delete image <name>
```

3. Provide users and admins a tool to display images available for job submission

```
vmgr show images
vmgr show image <name>
```

5.2.4 Integrated Components

There is a list of optional components that can be used with vTorque, but are not mandatory to have in place for it to work. vTorque works fine without any of them. However each of these MIKELANGELO components, described in the subsequent sections, has a clear purpose and comes with benefits that are worthy of consideration.

5.2.4.1 IOcm

IBM's improvements to the IO core manager for higher I/O throughput in KVM depends on a modified Linux 3.18 kernel and the dynamic I/O core manager. Dynamic core manager can be configured by vTorque specifying minimum and maximum number of cores the manager is allowed to consume. These parameters can be set by users (`vsub -vm iocm=true,iocm_mix_cores=1,iocm_max_cores=4`), once the feature is enabled by administrators (`IOCM_ENABLED`). Otherwise default values set by administrators will be applied (`IOCM_ENABLED_DEFAULT`, `IOCM_MIN_CORES_DEFAULT`, `IOCM_MAX_CORES_DEFAULT`).

5.2.4.2 ZeCoRX

At the time of writing there is no production ready version of IBM's zero-copy available yet, thus the integration is not done.

5.2.4.3 vRDMA Prototype 2 & 3

Huawei's first prototype of the vRDMA functionality was integrated into an earlier version of vTorque. There is an admin configuration option that controls if vRDMA is available at all to users (`VRDMA_ENDABLED`). The nodes that are equipped with vRDMA capable hardware need to be defined with a corresponding configuration option (`VRDMA_NODES`). Additionally there is also one default parameter (`VRDMA_ENABLED_DEFAULT`) to be defined by cluster admins, for the default behavior if there is no vRDMA parameter provided by the user at submission time (`vsub -vm vrdma=true`) and vRDMA is available and enabled.

The setup of the vRDMA bridge and its local DHCP server happens during the root prologue execution. The tear down and clean up are in the root epilogue. The integration of prototype 2 can be considered as tech preview, not ready for production. Integration of prototype 3 has not been started, due to required hardware not being available for our HPC development and testing environment.

5.2.4.4 UNCLOT

Shared memory component for intrahost VM communication can be enabled by administrators (`UNCLOT_ENBALED`), as well as the default behavior for vTorque (`UNCLOT_ENABLED_DEFAULT`). Furthermore, administrators can define the default amount of memory used for UNCLOT ivshmem (`UNCLOT_SHMEM`). Users can control the feature on the command line (`vsub -vm unclot=true,unclot_shmem=1024M jobScript.sh`)

5.2.4.5 Guest OS

The guest operating systems supported by vTorque are on the one hand standard linux guests (full OS) and on the other hand a stripped down lightweight unikernel OS optimized for clouds.

The only hard requirement is cloud-init[37], which allows a further customization of images during boot.

The environment differs between bare-metal and virtual nodes in a few aspects, these are:

- No Torque components installed in the guest OS
- MPI built without Torque support (Torque's tm.h header file required)
- PBS environment partially available
    - Basic ones available, i.e. `$PBS_JOBID, $PBS_NODEFILE`
    - Others skipped, i.e. `$PBS_MICFILE, $PBS_MOMPORT, $PBS_GPUFILE`

Standard Linux

Standard Linux guests (referred to as SLG) are supported by vTorque. To be more exact Debian based and RedHat based Linux distributions are supported.

MPI applications are started by the help of an mpi wrapper, defined in metadata templates created by cloud-init, which adds argument "`--hostfile <hostfile>`" to the `mpirun` command, if there is none of following arguments provided by the user:

```
-H|-host|--host|-hostfile|--hostfile-default-hostfile|--default-hostfile
```

OSv

OSv is the most lightweight option, with a small memory footprint and very short booting times. It comes with the benefit that it is executed in the userspace only, thus eliminating context switches and reducing the overhead, compared to standard Linux guests. The most remarkable difference is the lack of an SSH server: applications are started via RESTful interface:

```
curl -X PUT http://$FIRST_VM:8000/app/ --data-urlencode
command="$cmd"
```

and require polling for status checks:

```
curl --connect-timeout 2 \
          -X GET http://$FIRST_VM:8000/app/finished \
          --data-urlencode tid="$tid"
```

While applications run in SLG, started via SSH, they block until the program finishes and an exit code is returned. In OSv there is polling required to check if the application is still active and to fetch its exit code. Further details about OSv can be found in D2.21 "MIKELANGELO Final Architecture" and D6.2 "Final report on the Architecture and Implementation Evaluation".

5.2.4.6 LEET

The packaging tool LEET is not directly integrated into vTorque. It is intended to be available to admins and developers, only, those who need to package applications with OSv, not for regular users. Thus there is no point in direct vTorque integration with LEET as an optional component available cluster wide, instead it is recommended to install it on a dedicated build host. Virtual machine images composed with LEET are directly usable with vTorque.

5.2.4.7 Snap Telemetry

Snap-Telemetry and monitoring is recommended for vTorque deployments in general. Its installation is straightforward and as it doesn't have a noticeable overhead, but on the other hand provides valuable insights into your resource utilizations and helps to identify bottlenecks in the infrastructure and application performance, there is no point in skipping it.

In the third year of the project updates for Snap have been deployed, but since its interfaces and configuration didn't change, there was no need to update its integration into vTorque.

5.3 vTorque Setup

To provide administrators with an easy and convenient way to setup vTorque on top of an existing Torque, a setup script written in bash is provided. However, administrators can install it also manually. All steps are outlined in the subsequent section.

All requirements for vTorque concerning software and infrastructure need to be fulfilled as outlined in section 5.1.1. Additionally, the person taking care of the setup needs to have root access in the cluster environment.

5.3.1 Setup Script

The setup script supports both an installation and removal of a vTorque deployment. The following arguments are accepted:

`-u|--usage`
: Prints usage.

`-p|--prefix`
: Installation prefix, destination directory is $PREFIX/vtorque.

`-u|--uninstall`
: Remove vTorque, be careful to provide argument '`-p`' if a custom installation path was used.

To install vTorque in its default location `/opt/vtorque`, simply run the script without any arguments.

```
my_host:/tmp/vTorque# setup.sh
```

If you want to have it installed somewhere else, issue one of:

```
setup.sh --prefix /another_dir_than_opt
setup.sh -p /another_dir_than_opt
```

During installation you will be asked for a few configuration settings, namely network settings applied to VMs. These settings include:

- Domain: your domain, i.e. `department.mycompany.com`
- Search Domain: name server, i.e. `department.mycompany.com`
- NTP server 1+2: the NTP servers to use for time synchronisation, i.e. `ntp{1,2}.mycompany.com`
- NFS Server: shared storage for your cluster, i.e. `nfs.intranet`

For a full list of configuration options, please refer to section 5.4.

5.3.2 Manual Installation

Administrators do not need to make use of the provided setup script, as it just requires a few steps that can be done manually in a timely manner. A base path for the target installation `/opt` is used as an example. In general vTorque needs to be available on the submission front-end as well as on the compute nodes. The following how-to lists each step:

Deploy vTorque on shared storage, i.e. in `/opt/vtorque`, accessible from front-end and compute nodes. If there is a dedicated cluster head node where the scheduler runs, access from it to the shared storage is not required.
Switch into the cloned vTorque directory, and copy files into the destination directory:

```
DEST_DIR=/opt/vtorque;
cp -r ./lib $DEST_DIR/;
cp -r ./src/* $DEST_DIR/;
cp -r ./doc $DEST_DIR/;
cp -r ./test $DEST_DIR/;
cp ./LICENSE $DEST_DIR/;
cp ./NOTICE $DEST_DIR/;
cp ./README* $DEST_DIR/;
```

Adopt relative paths

```
sed -i 's,../src/,../,g' $DEST_DIR/doc/*;
```

Deploy profile on all front-end nodes:

```
cp ./contrib/99-mikelangelo-hpc_stack.sh /etc/profile.d/;
```

1. Add vTorque installation directory `/opt/vtorque` to `$PATH` on all nodes that require to be able to submit jobs

```
sed -i -e "s,VTORQUE_DIR=.*,VTORQUE_DIR=\"$DEST_DIR\";,g"
/etc/profile.d/99-mikelangelo-hpc_stack.sh;
```

2. Setup vTorque on compute nodes

```
cp ./contrib/99-mikelangelo-hpc_stack.sh /etc/profile.d/;

DEST_DIR=/opt/vtorque;
sed -i -e \"s,VTORQUE_DIR=.*,VTORQUE_DIR=$DEST_DIR,g\"
               /etc/profile.d/99-mikelangelo-hpc_stack.sh;

rename -v 's/(.*)\$$$\/\$$$\1.orig/' \
               /var/spool/torque/mom_priv/\
                   {pro,epi}logue{.user,}{.parallel,.precancel,};

ln -sf $DEST_DIR/src/scripts/\
           {prologue{,.parallel},epilogue{,.parallel,.precancel}}\
```

```
/var/spool/torque/mom_priv/;";
```

3. Ensure correct permissions for vTorque (once, only, since dir is shared amongst nodes)

```
DEST_DIR=/opt/vtorque;
chown -R root:root $DEST_DIR;
chmod -R 555 $DEST_DIR;
chmod 444 $DEST_DIR/contrib/*;
chmod 444 $DEST_DIR/doc/*.md;
chmod 444 $DEST_DIR/src/common/*;
chmod 500 $DEST_DIR/src/scripts/*;
chmod 500 $DEST_DIR/src/scripts-vm/*;
chmod 444 $DEST_DIR/src/templates/*;
chmod 444 $DEST_DIR/src/templates-vm/*;
```

4. Configure vTorque, check file `/opt/vtorque/common/config.sh`. For a full list of configuration options, valid values and description of options, please refer to section 5.4.

5.4 Infrastructure Environment Configuration

There are reasonable defaults set whenever possible, however administrators need to check a few options (i.e. path to shared storage, NFS server, NTP servers, etc) after installation. It is recommended to go through the whole configuration file `common/config.sh` and check each option.

The following subsection highlights all newly introduced and modified configuration options, while the complete lists can be found in Appendix A.

5.4.1 New Configuration Options for Cluster Administrators

Table 6 below lists all newly introduced configuration options that enable administrators to control which vTorque features and optional components are made available to users. A full list can be found in Appendix A.1.

Table 6: New vTorque configuration options

Parameter	Expected Values	Description
`UNCLOT_ENABLED`	`true/false`	Dis/enables UNCLOT component.
`NFS_TIMEOUT`	`in seconds`	Time out for flat file flags to appear on remote nodes (e.g. NFS shared home)

MEASURE_TIME	true/false	Measures execution time of all scripts involved in a job's life cycle.

5.4.2 New Defaults Configuration for Cluster Administrators

Table 7 below lists all newly introduced default configuration options that enable administrators to provide meaningful defaults for all vsub command line options, enabled in the configuration explained in section 5.4.1. A full list can be found in appendix A.2

Table 7: New vTorque configuration option defaults

Option	Expected value(s)	Description
UNCLOT_ENABLED_DEFAULT	true/false	Dis/enables UNCLOT component per default, if user does not provide the corresponding argument.
UNCLOT_SHMEM_DEFAULT	in K/M/G/T	Amount of memory for UNCLOT.

5.4.3 New Command Line options for vsub Command and job script inline Options

All vsub command line options, related to the virtualized execution of batch jobs, are prefixed by '-vm' followed by a comma separated list of options. There is only one exceptions, the parameter '-gf' that prevents the actual submission, but generates all wrapper scripts for manual execution, very beneficial for debugging vTorque. This needs to be enabled by administrators before invoking, as it may allow users to exploit the cluster.

Standard Torque's qsub command line for the submission of batch job scripts looks like this:

```
qsub [-l standard_pbs_parameters] <job script>
```

e.g. `qsub -l nodes=4 myJob.sh`

The extended command line for vTorque's vsub offers additional parameters for virtualization:

```
vsub [-gf] [-vm vm_parameters] [-l standard_pbs_parameters] <job script>
```

e.g. `vsub -vm img=ubuntu.16_04-LTS.img -l nodes=1 jobScript.sh`

Parameters for virtual guests (the `[-vm <vm_parameters>])` are a list of comma separated "key=value" pairs. The table below lists all new command line options available for vsub.

Table 8: New vTorque command line options

Argument	Expected value(s)	Short Description
`unclot`	`enabled|true|yes|0|disabled|false|no|1`	Dis/enable UNCLOT for OSv guests.
`unclot_shmem`	`in K/M/G/T`	Amount of memory for UNCLOT.

5.5 CI Integration with Scotty

The CI integration with Scotty enables the running of a predefined list of different experiment workloads in an automated manner, including collecting and archiving of performance measurement results.

It enables use case providers to analyse the behavior of their specific HPC use case with the newly developed components and on the other hand to identify opportunities for further improvements of the cross-layer optimization components provided by MIKELANGELO.

5.5.1 Setup

For a successful Scotty setup utilizing the HPC workload generator, responsible for executing experiment workloads on HPC backends, 3 things need to be in place:

- Scotty installation (for details please refer to 3.3)
- Experiment workload definitions (i.e. `Bones_UC-CI_Experiment[38]`), which may be hosted locally or remotely
- Access to the hpc-workload-gen project, referenced in the experiment workload definitions.

5.5.2 HPC Workload Gen

The HPC Workload generator (`hpc-workload-gen[39]`) is a middleware between CI systems like Jenkins or Gitlab CI and an HPC environment with PBS Torque batch system manager available. The aim of this project is to implement for Scotty (see section 3.3) an HPC backend that can be used to execute experiments also on an HPC backend, including support for vTorque. Details are outlined below.

HPC backend Config

The HPC backend configuration file for the hpc-workload-gen is called `hpc_backend.cfg` and is to be found in the git. It contains placeholders, which are replaced during execution by

the hpc-workload-gen if found in the environment. You may instead clone the repo, edit the file hpc_backend.cfg and refer in your yaml file to the cloned one with those details instead. However it is recommended to provide them via the environment.

Table 9 below gives an overview about all HPC backend configuration options. Environment variables are expected to be named exactly as their corresponding configuration option, keep in mind the options are case sensitive.

Table 9: All HPC backend configuration options

Configuration Option	Expected value	Description
`domain`	FQDN	Used to identify jobs, since they have by default the domain name included.
`host`	FQDN	Submission frontend reachable via SSH.
`user_name`	user name	SSH remote username of the frontend.
`ssh_port`	1-65535	Port of the remote SSH server on the frontend.
`ssh_key`	absolute path	Path to ssh key for user on frontend.
`grafana`	FQDN	URL of the grafana dashboard.
`grafana_dashbord_name`	name	Name of the dashboard displaying the live measurements.
`grafana_host`	FQDN	Hostname of the grafana dashboard.
`path_qstat`	absolute path	Path to Torque's qstat on the frontend.
`path_qsub`	absolute path	Path to Torque's qsub on the frontend.
`path_vsub`	absolute path	Path to vTorque's vsub on the frontend.
`path_vtorque_log`	path	Pathprefix (jobID is needed) to path to job's vTorque log.
`execution_dir`	path	Remote dir on frontend where the job will be submitted from.
`poll_time_qstat`	seconds	Time span between polling if a job has been completed, the lower the time

		span the higher the load for the batch system.

Experiments Definition

Automated experiment execution requires a definition of each experiment to be executed. These are, just as the case for the Cloud counterpart (see section 3.3), a YAML file processed by Scotty steering the experiment workload generator, hpc-workload-gen.

Below is an example of a YAML file for the Cancellous Bones HPC Use Case.

```
description: Cancellous Bones Use Case
# for scotty, describes experiment
tags:
  - hpc
  - mpi
  - bones_uc

workloads:
#
# experiment configuration #1 - Bare metal
#  -bare metal
#  -one of two sockets
#
  - name: bones_uc-bare_metal_numa_node
    generator: git:ssh://git@gitlab.xlab.si:13022/mikelangelo/hpc-
workload-gen.git
    params:
      hpc_config: hpc_backend.cfg
      job_script: job_scripts/run_bones_uc.sh
      input_data: bones_uc/input_data/
      qsub_args: "-l nodes=1"
      vsub_args: "-vm vcpus=8 -vm ram=8012M"
#
# two sockets
#
  - name: bones_uc-bare_metal_1_node
    generator: git:ssh://git@gitlab.xlab.si:13022/mikelangelo/hpc-
workload-gen.git
    params:
      hpc_config: hpc_backend.cfg
      job_script: job_scripts/run_bones_uc.sh
      input_data: bones_uc/input_data/
      qsub_args: "-l nodes=1"
      vsub_args: "-vm vcpus=16 -vm ram=8012M"
#
# ...
#
```

5.6 Conclusions and Future Work

vTorque has reached a stable and feature-rich state. It is capable of executing virtualized workloads in Torque-based HPC batch systems. It comes with a lot of functionality targeting cross-layer improvements throughout the whole software stack with respect to administrator and end user's experience.

Nevertheless not all features and optional nice-to-have functionality have been accomplished, and is summarized as future work for the following areas:

- I/O improvements to lower the virtualization overhead further
  - ZeCoRX
  - vRDMA Prototype 3
- Resilience and fault tolerance
  - Live migration in combination with global spare nodes (patch Torque)
  - Suspend+resume support for virtual guests in the vTorque layer
  - Introduce ability to support suspension and resumption in application layers (MPI OpenMP, …) running in VMs
- Increase user experience
  - Implement vsub in c/c++ (increase job throughput)
  - Implement vmgr in c/c++ (faster response times)
  - Replace flat file check in root-prologue (adds few seconds overhead) if it is a VM job, by a trigger mechanism instead
- Introduction of new features
  - Interactive VM jobs, useful for remote visualization
  - Support for more file-systems in OSv (Lustre, BeeGFS, ..)

6 Key Takeaways

The key takeaways of this deliverable are:

- The full-stack Instrumentation and Monitoring system can now collect data from KVM (and sKVM), and OpenVSwitch. Performance, functionality and robustness improvements have been carried out on all previously contributed plugins. Tools to simplify installation and configuration have been developed. All contributions have been open-sourced.
- The Lightweight Execution Environment Toolbox (LEET) now supports the ability to deploy OSv-based unikernels in parallel with Docker containers on Kubernetes environments. Numerous contributions have been made to Virtlet and it has been demonstrated successfully hosting OpenFOAM, Apache Spark and microservice workloads.
- The Scotty integrated experimentation workflow has been simplified and reimplemented to deliver a streamlined, extensible continuous experimentation framework. Experiment configuration is now via a single yaml file, and the complete lifecycle up to and including data analysis across experiment runs is now automated.
- A highly scalable and extensible Actuator framework has been implemented that allows arbitrary systems to be manipulated via relevant plugins. This provides a consistent and automated way to configure or reconfigure arbitrary software components across a highly distributed workload deployment.
- The MIKELANGELO Cloud Manager has been rewritten and extended to allow live resource scheduling algorithms to be explored on top of infrastructures including OpenStack and snap. It has been integrated with both Scotty and Actuator.py, allowing algorithms to be manipulated in continuous experimentation configuration, and to manipulate arbitrary software components in the deployment, as required.
- The Cloud testbed constructed by year two of the project has been enhanced with further integration of MIKELANGELO components as they have become available. IOcm, Snap, SCAM, OSv, LEET, Scotty, Actuator and MCM are all now successfully integrated, and the testbed is now hosting a range of workloads and use cases
- The physical HPC infrastructure provisioned by the end of Year two has continued to support the needs of the project. All supporting services have been virtualised, enhancing both reconfigurability and redundancy. A refined development workflow has been deployed, isolating long term public repositories of stable code from temporary development repositories. Dedicated vsub and vmgr cli tools have been developed for improved performance. Numerous MIKELANGELO components have been successfully integrated, including Scotty enabling automated execution of multiple workloads.

7 Concluding Remarks

The MIKELANGELO project has completed its three year programme of work. Comprehensive and innovative tools to enable the integration of the complete MIKELANGELO stack have been developed and two production-mirroring testbeds have been commissioned - one for Cloud and one for HPC.

Enabling tools include a scalable full-stack instrumentation and monitoring solution which has been fully open-sourced and has been enhanced to deliver functionality, manageability and performance improvements. The Lightweight Execution Environment Toolbox, and the third-party Virtlet project, have been updated to now allow OSv-based unikernels to be hosted on the popular Kubernetes container management platform.

The Scotty continuous experimentation framework has been simplified and reimplemented, and supports the complete experiment lifecycle up to and including statistical data analysis across experimentation runs. A scalable and highly extensible Actuator framework has been developed to manipulate arbitrary resources that are required by a workload. The MIKELANGELO Cloud Manager has been rewritten and extended to allow live resource scheduling algorithms to be explored on top of infrastructures including OpenStack and snap.

Regarding Cloud integration, the Cloud testbed has been enhanced with further integration of MIKELANGELO components. The latest versions of IOcm, Snap, SCAM, OSv, LEET, Scotty, Actuator and MCM are all now successfully integrated.

Regarding HPC integration, the vTorque virtualisation layer for HPC has matured and dedicated command line tools have been delivered. The HPC management layer is now virtualised and an enhanced automated workflow for deploying HPC workloads on virtualised infrastructure has been implemented. Comprehensive integration with Scotty has been demonstrated..

MIKELANGELO has thus developed a rich and comprehensive suite of tools that can be used independently or together to construct highly integrated, distributed, scalable testbeds. These tools have enabled two full-stack deployments, one for Cloud and one for virtualised HPC, that have been used to host and prove the various performance and security enhancing components developed by MIKELANGELO, and exercised by the MIKELANGELO use-cases.

8 Appendix A - vTorque Configuration and Options

This appendix lists all configuration options for cluster administrators, as well as the full list of command line options, which can also be used as inline options within job-scripts.

8.1 Configuration Options for Cluster Administrators

In the following table all configuration options to be found in vTorque's configuration file common/config.sh are listed and described.

Table 10: All vTorque configuration options

Parameter	Expected Values	Description
DISABLE_MIKELANGELO_HPCSTACK	true|0|false|1	Disable/enable vTorque job submission.
DISABLED_HOSTS_LIST	regular expression	Hostnames that cannot use vTorque functionality, but Torque only.
PARALLEL	true|0|false|1	Execute guest boot processes on the remote nodes asynchronously or sequentially.
ALLOW_USER_IMAGES	true|0|false|1	Indicates whether users are allowed to submit images with their job that are not stored in dir 'IMAGE_POOL'
IMAGE_POOL	<abs. dir path>	Image pool dir for (authorized/verified) images.
HOST_OS_CORE_COUNT	0..<max avail>	Count of cores dedicated to the host OS.
HOST_OS_RAM_MB	0..<max avail>	Amount of RAM in MB dedicated to the host OS.
MAX_VMS_PER_NODE	1..<max>	Maximum count of VMs per node that cannot be exceeded.
STATIC_IP_MAPPING	true|0|false|1	Use a static mapping of MAC addresses to IP addresses, instead of a DHCP server. Requires implementation of a custom

		mapping function.
TIMEOUT	0..<max>	Timeout in seconds for remote processes to complete booting or destruction of VMs. Must be lower than pbs_mom's timeout for pro/epilogue.
SERVER_HOSTNAME	regular expression	Regular expression matching the frontend Hostnames. Used to determine the path of PBS qsub.
REAL_QSUB_ON_SERVER	<abs. dir path>	Absolute path to PBS qsub command on the frontend.
REAL_QSUB_ON_NODES	<abs. dir path>	Absolute path to PBS qsub command on the compute nodes
IOCM_ENABLED	true|0|false|1	Dis/enable IOCM.
IOCM_MIN_CORES	0..<max avail>	Min number of cpus that are reserved for IOCM.
IOCM_MAX_CORES	1..<max avail>	Max number of cpus that are reserved for IOCM.
VRDMA_ENABLED	true|0|false|1	Dis/enable vRDMA.
VRDMA_NODES	regular expression	Hostnames with vRDMA capable hardware.
SNAP_MONITORING_ENABLED	true|0|false|1	Dis/enable tagging of jobs.
UNCLOT_ENABLED	true/false	Dis/enables UNCLOT (isvhmem OSv) component.
NFS_TIMEOUT	in seconds	Time out for flat file flags to appear on remote nodes (e.g. NFS shared home).
MEASURE_TIME	true/false	Measures execution time of all scripts involved in a job's life cycle.

8.2 Configuration Option Defaults for Cluster Administrators

In the following table all default configuration options for vTorque's job execution are listed. They can be overridden by users on the command line, as long as the feature to override its default is enabled by admins in the first place. These defaults are intended to provide meaningful settings to their users for all cases where nothing is provided on either the command line, or inline within the job script.

Table 11: All vTorque configuration option defaults

Option	Expected value(s)	Description
FILESYSTEM_TYPE_DEFAULT	Shared file system 'FILESYSTEM_TYPE_SFS' and RAM disk 'FILESYSTEM_TYPE_RD' are accepted.	Controls where the image is copied to for execution. Either to a local RAM disk or to the shared file-system.
IMG_DEFAULT	*.img|*.qcow2	Default image for virtual guests.
DISTRO_DEFAULT	debian|ubuntu|redhat|centos|fedora|osv	Distro of the guest's image, depends on the default image.
ARCH_DEFAULT	X86_64|x86_i386|arm	Architecture of the default image. Usually 'x86_64'
VCPU_PINNING_DEFAULT	true|0|false|1	Pin virtual cpus, recommended to be enabled.
VCPUS_DEFAULT	0..<max cores avail>	Default amount of virtual CPUs per guest.
RAM_DEFAULT	1..<max avail per VM>	Default amount of RAM in MB dedicated to each virtual guest.
VMS_PER_NODE_DEFAULT	1..<max possible>	Recommended value is '1'. Must be greater or equal to 1.
DISK_DEFAULT	*.img|*.qcow2	Optional persistent user disk, mounted in the first VM.
HYPERVISOR_DEFAULT	kvm|skvm	Default hypervisor for guests. Note: IOcm requires sKVM.
VRDMA_ENABLED_DEFAULT	true|0|false|1	Will be ignored if config option VRDMA_ENABLED is set to false.

IOCM_ENABLED_DEFAULT	true|0|false|1	Will be ignored if config option IOCM_ENABLED is set to false.
IOCM_MIN_CORES_DEFAULT	1..<max avail>	Recommended value is '1'. Must be greater or equals 1.
IOCM_MAX_CORES_DEFAULT	1..<max avail>	Must be greater or equals IOCM_MIN_CORES_DEFAULT.
UNCLOT_ENABLED_DEFAULT	true/false	Dis/enables UNCLOT component per default, if user does not provide the corresponding argument.
UNCLOT_SHMEM_DEFAULT	in K/M/G/T	Amount of memory for UNCLOT.

8.3 Command Line Options for vsub Command and job script Inline Options

All options listed in the table below can either be issued on the vsub command line or prefix by '#VPBS ..', similar to the way Torque accepts options using '#PBS ..'.

Table 12: All vTorque command line options

Argument	Expected value(s)	Short Description	State
-gf	n/a	Not prefixed by '-vm' and not relevant for bare metal jobs. It causes to generate files, but to not submit them. Useful for debugging, only.	*n*
img	*.img|*.qcow2	Image file for the virtual guest. Relative paths are looked up in IMAGE_POOL_DIR.	-
distro	debian|ubuntu|redhat|centos|fedora|osv	Distro of the image. Can be skipped if the image file name contains it.	-
ram	0..<max avail>	Amount of RAM in MB dedicated to each virtual guest.	-
vcpus	1..<max avail>	Amount of cores dedicated/pinned to each virtual guest.	-

vms_per_node	1..<max>	Amount of virtual guests per host.	*n*
disk	*.img|*.qcow2	Optional persistent user disk that is mounted to the first guest.	*n*
arch	X86_64|x86_i386|arm	Architecture of the guest image.	-
hypervisor	kvm|skvm	Hypervisor to use for the virtual guests.	-
vcpu_pinning	true|0|false|1	Pin virtual cores to physical cores.	*u*
vm_prologue	<abs. file path>	Optional user prologue, executed in the virtual guest before the job script.	*n*
vm_epilogue	<abs. file path>	Optional user epilogue, executed in the virtual guest before the job script.	*n*
vrdma	true|0|false|1	Dis/enable vRDMA.	*n*
iocm	true|0|false|1	Dis/enable IOCM.	*n*
iocm_min_cores	0..<max avail>	Skipped if `iocm` is false.	*n*
iocm_max_cores	1..<max avail>	Skipped if `iocm` is false.	*n*
fs_type	sharedfs|ramdisk	Filesystem to use for copying the image file for execution. One copy per VM.	*n*
unclot	enabled|true|yes|0|disabled|false|no|1	Dis/enable UNCLOT for OSv guests.	y
unclot_shmem	in K/M/G/T	Amount of memory for UNCLOT.	1G

9 References and Applicable Documents

[1] MIKELANGELO Report D2.21 The final MIKELANGELO architecture. https://www.mikelangelo-project.eu/wp-content/uploads/2017/09/MIKELANGELO-WP2.21-USTUTT-v2.0.pdf
[2] MIKELANGELO Report D5.7 https://www.mikelangelo-project.eu/wp-content/uploads/2016/06/MIKELANGELO-WP5.7-Intel-DE_v2.0.pdf
[3] MIKELANGELO Report D5.2 Intermediate report on the Integration of sKVM and OSv with Cloud and HPC. https://www.mikelangelo-project.eu/wp-content/uploads/2017/01/MIKELANGELO-WP5.2-INTEL-v2.0.pdf
[4] Snap plugin catalogue, http://snap-telemetry.io/plugins.html
[5] Snap plugin repositories, https://github.com/intelsdi-x/snap/blob/master/docs/PLUGIN_CATALOG.md
[6]KVM Hypervisor, available at https://www.linux-kvm.org/
[7]OpenStack, available at https://www.openstack.org/
[8]vTorque, available at https://www.mikelangelo-project.eu/technology/vtorque-virtualization-support-for-torque/
[9]Snap libvirt Collector Plugin, available at https://github.com/intelsdi-x/snap-plugin-collector-libvirt
[10] Snap deploy application, https://github.com/intelsdi-x/snap-deploy
[11] OpenStack Keystone Identity Service, available at https://docs.openstack.org/keystone/latest/
[12] UniK, available at https://github.com/solo-io/unik
[13] EMC releases UniK, available at https://www.emc.com/about/news/press/2016/20160524-01.htm
[14] Kubernetes CRI, http://blog.kubernetes.io/2016/12/container-runtime-interface-cri-in-kubernetes.html
[15] Virtlet GIthub Project, https://github.com/mirantis/virtlet
[16] https://github.com/Mirantis/virtlet/blob/master/docs/architecture.md
[17] https://github.com/mikelangelo-project/osv-openfoam-demo
[18] https://github.com/mikelangelo-project/osv-spark-demo
[19] https://www.mikelangelo-project.eu/2017/05/the-microservice-demo-application-introduction
[20] HiBench, https://github.com/intel-hadoop/HiBench
[21] https://about.gitlab.com/
[22] https://docs.openstack.org/nova/latest/
[23] https://docs.openstack.org/heat/latest/
[24] https://www.influxdata.com/
[25] https://www.mongodb.com/
[26] https://about.gitlab.com/features/gitlab-ci-cd/
[27] https://docs.gitlab.com/runner/
[28] https://linux.die.net/man/1/taskset
[29] http://kernel.ubuntu.com/~cking/stress-ng/

[30] https://zookeeper.apache.org/
[31] MIKELANGELO Report D2.19 The first MIKELANGELO architecture.
http://www.mikelangelo-project.eu/deliverables/deliverable-d2-19/
[32] http://arc-ts.umich.edu/software/torque/
[33] https://www.gerritcodereview.com/
[34] https://fusionforge.org/
[35] https://github.com/
[36] MIKELANGELO Report D2.20 The intermediate MIKELANGELO architecture.
https://www.mikelangelo-project.eu/wp-content/uploads/2016/07/MIKELANGELO-WP2.20-USTUTT-v2.0.pdf
[37] https://cloud-init.io/
[38] https://github.com/mikelangelo-project/Bones_UC-CI_Experiment
[39] https://github.com/mikelangelo-project/hpc-workload-gen