MIKELANGELO

D5.1

First Report on the Integration of sKVM and OSv with Cloud Computing

Workpackage	5	Infrastructure Integration	
Author(s)	Peter Chronz		GWDG
	Maik Srba		GWDG
	Gregor Berginc		XLAB
	John Kennedy		INTEL
Reviewer	John Kennedy		INTEL
Reviewer	Michael Gienger		HLRS
Dissemination Level	PU		

Date	Author	Comments	Version	Status
2015-12-13	Peter Chronz	Initial draft	V0.1	Draft
2015-12-13	Gregor Berginc	Integration of MPM	V0.2	Draft
2015-12-14	John Kennedy	Monitoring content added	V0.3	Draft
2015-12-22	Peter Chronz	Draft ready for review	V1.0	Draft
2015-12-28	Peter Chronz	Document ready for submission	V2.0	Final

Executive Summary

This report describes the initial integration of the MIKELANGELO components for virtualisation and cloud computing within an unified cloud computing architecture. While other parts of the project work on the underlying technologies such as the hypervisor and the guest OS, this workpackage leverage those components to improve cloud computing.

The approach in this workpackage is to integrate the results regarding virtual infrastructure technology with service management tools in a dominant cloud stack. This report is analogous to the integration of the MIKELANGELO components with HPC. Since this is the first report we first define our goals and devise a work plan. We then discuss how individual components can be used to reach those goals. Furthermore, we research options for a cloud computing framework as basis for integration. Another concern of our work is the quality assurance, which receives special attention in such a large project. Spanning a range from the host kernel to big data applications running a virtualised cluster, the cloud integration becomes a major feat in systems engineering. Thus, we dedicate a significant portion of our work and this report to setting up continuous integration.

Being the first iteration of this report, after only six month of work, this report contains a high ratio of conceptual work. However, we also describe concrete technical work being carried out for the demonstrator at the beginning of the second project year. This demonstrator will integrate key components of MIKELANGELO in a cloud computing framework for the first time. The concrete results include the choice of OpenStack as cloud stack, the integration of MPM for application management, Snap for monitoring, and initial results of deploying a working testbed with continuous integration.

We conclude this report by giving an outlook onto future iterations of this work and our next steps. This outlook includes the setup of a project-wide testbed, integration with Jenkins for continuous integration, and making the testbed available to implement and execute use cases. Furthermore, we will investigate the use of further container-based technologies such as Kubernetes and alternative guest OSs for the cloud such as CoreOS.

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)

Table of Contents

List of Figures

List of Abbreviations

CLI	Command Line Interface
DHCP	Dynamic Host Configuration Protocol
HPC	High Performance Computing
IOcm	I/O Core Manager
KVM	Kernel-based Virtual Machine
MPM	MIKELANGELO Package Manager
NIC	Network Interface Card
OS	Operating System
RAID	Redundant Array of Inexpensive Disks
RDMA	Remote Direct Memory Access
RoCE	RDMA over Ethernet
SAN	Storage Area Network
SCAM	Side-Channel Attack Monitoring and Mitigation
sKVM	super KVM
VM	Virtual Machine
VMI	Virtual Machine Image
vRDMA	virtual RDMA
VXLAN	Virtual Extensible LAN

1 Introduction

This report describes the initial integration of the MIKELANGELO stack with cloud computing. As the initial report of the cloud integration this report describes the initial work plan and the work carried out in the second half of the first project year.

Cloud computing builds heavily on virtualisation. Nearly all deployments of cloud computing nowadays use some kind of virtualisation. Traditionally mostly full virtualisation such as that by KVM, Xen or VMWare is used. The recent trends show an increased enthusiasm for container-based virtualisation. Although these types of virtualisation have some inherent drawbacks, such as lack of snapshotting, a static kernel, and no live-migration they enjoy popularity. There are two main reasons for this popularity and for the gains in market share and build-up hype that surrounds projects such as Docker and Kubernetes. The first reason is the simple management of services. Here Docker and Kubernetes hit a sweet spot between IaaS and PaaS. The other reason for the popularity of container-based virtualisation is their performance. Containers run in isolation on the native system with very little overhead. This gives them their high performance. However when investigating performance further, we see that full virtualisation technologies reach similar performance for compute operations. It's only I/O that is significantly faster with containers that VMs.

MIKELANGELO works to overcome the mentioned disadvantages in management and performance for full virtualisation. At the same time we will retain their benefits such as snapshotting, portability, and live-migration. The first aspect, improved service management will emulate the service deployment and management model of Docker. The central element of this aspect is our application management service called MPM. The second aspect, performance, will be tackled by improving KVM and by using and improving OSv [33]. The work described in this report revolves around the integration of those technologies with cloud computing. In this work, we leverage the results of MIKELANGELO to allow provide the same benefits that Docker provides in conjunction with full virtualisation and proven cloud computing stacks.

The main challenges of this work revolve around the management and extension of complex cloud stack, integration with those cloud stack, and cross-layer optimisation. After approximately ten years of research cloud computing is maturing. The major players have gained major market shares and few mature stacks are developed with high momentum. This also leads to an increased complexity of the cloud computing stacks. While cloud computing stacks more features required for production deployments, their deployment and maintenance becomes also more complicated. The same is true for their development.

Providing a deep and meaningful integration of MIKELANGELO with a dominant cloud computing stack that can be upstreamed and that will provide clear added value is our main challenge. While being a serious challenge, we do not foresee it as unreasonably hard to achieve.

In this document, we present a work plan, the architecture of the MIKELANGELO cloud stack, our approach to quality assurance, and the first work on an integrated demonstrator. Our approach is to communicate communicate clearly with all work packages, to steer development such that it can be integrated well with cloud computing. This involves feedback loops with work packages 2, 3, and 4. In addition, the work in this workpackage will be the basis for the work carried out in workpackage 6. To ensure a smooth integration across the whole stack quality assurance is a major concern. We tackle this aspect by employing a continuous integration framework. The preliminary results of the integration work is also described in this document. These results contain descriptions of a productive cloud testbed that is being set up and the initial experiences made with integrating MIKELANGELO components according to our cloud architecture. Most concrete results are reflected in the description of the demonstrator of the cloud stack.

As of writing this report conceptual work has been performed and implementation work is undergoing. Specifically the cloud architecture has been researched and designed. A comparison of cloud middleware has been conducted and is presented in this document. The implementation work includes the setup of a testbed for a production-grade cloud based on OpenStack. The integration of application management with MPM, based on OSv has been carried out with initial results. The integration of holistic monitoring via implementation of initial snap plugins has been carried out as well. The initial release of the continuous integration service is being prepared and expected for M13 (January 2016). Finally, the implementation of the first demonstrator on basis of the cloud integration is being carried out currently, with expected delivery in M14 (February 2016).

This document is structured as follows. Section 2 describes our work plan including goals and an implementation strategy. Section 3 describes the integration of major components of the MIKELANGELO stack. Section 4 describes our approach to quality control via continuous integration. Section 5 describes the first demonstrator to be implemented at the beginning of the second project year. Section 6 concludes the initial integration work with an outlook.

2 Planning the Cloud Integration

The work plan for the integration of the MIKELANGELO components with cloud computing focuses on three major areas in year 1: the goals of the cloud integration, the implementation strategy, and roadmap including major milestones.

2.1 Goals of the Cloud Integration

From a high level viewpoint the two major goals of the cloud integration are

1. the integration of all separate MIKELANGELO components with a cloud middleware and
2. providing a basis for further research within MIKELANGELO and beyond.

2.1.1 Architecture Integration

The first high level goal, the integration of all MIKELANGELO components related to cloud computing, has the three goals of ***integration, verification, and sustainability***.

Integration refers to providing a cloud stack that combines all of MIKELANGELO's components that relate to cloud computing. The only exception here is work related to Infiniband, which is not considered to be central to typical clouds[1]. The integration with cloud computing also relates to the implementation of specific components for application packaging and management. Furthermore, the research and implementation of resource optimisation algorithms for I/O metrics is considered part of the cloud integration as well.

Verification refers to testing the components in an integrated scenario, providing continuous integration to test combinations of components, and to execute scenarios from the use cases for verification. The high level tests that come from use cases can be used especially to verify the functionality of package management, service management, and the system performance. An important aspect of performance verification requires to provide comparisons to cases that do not include modified MIKELANGELO components.

Sustainability refers to preparations to ensure the continued use and development of our outputs in the field of cloud computing beyond the project's runtime. In this project, we aim to upstream as many of our results as possible. Thus, we hope to achieve a great sustainability of our project results. The consortium has been constructed in a way to ensure

[1] Although some providers offer HPC clouds with Infiniband, we expect Converged Ethernet to remain the standard interconnect in clouds for very most applications in the long term.

a high degree of upstreaming since our consortium members are active participants in relevant developer communities.

2.1.2 Research Basis

The second high-level goal, providing a basis for research, has four related aspects: optimisation for I/O, hardening security, optimisation for cloud bursting, and providing comparisons.

The optimisation for I/O needs to be adaptable to satisfy varying demands for I/O performance for metrics such as aggregate I/O throughput, individual aggregate throughput, and I/O latency. This I/O optimisation will happen by leveraging the I/O capabilities and signalling from sKVM. The goal at the cloud layer is to optimise I/O metrics according to demands across Virtual Machines (VMs) and hosts. While early algorithms should only ensure a fair distribution of I/O performance, later algorithms should also allow to provide different levels of quality.

Hardening security refers to the successful integration of security mechanisms developed in the context of sKVM. Although sKVM can spot malicious VMs via the SCAM (Side Channel Attack Monitoring and Mitigation) component [1], the mechanisms need to be incorporated into the cloud deployment for realistic use. In specific, signals from SCAM about malicious VMs needs to be handled by the cloud layer to suspend and isolate malicious VMs, to allow for forensics, and to deal with their owners accordingly.

Optimisation for cloud bursting refers to the use of OSv with slim images and fast boot times. In addition, we will integrate this work with the cloud layer. The goal in the cloud layer is to set up new service instances in the cloud on appropriate hosts to allow for the best way to offer new instances quickly, boot up many new instances, and to deal with the growing I/O demand gracefully.

Providing comparisons refers to the goal of providing comparative results for functionality and performance. We can only know how our new work performs in the cloud if we can compare it to the legacy cloud. These comparisons need to include an upper boundary, which shall represent the best-possible scenario by using either native or container-based deployment. Furthermore, these comparisons need to include a legacy cloud deployment with standard components such as Ubuntu guests with KVM and un-modified cloud middleware.

2.2 Strategy

The implementation strategy parallels the structure of the goals in the previous section. The structure distinguishes two major accomplishments: architecture integration and providing a basis for research.

2.2.1 Architecture Integration

The architecture integration itself is divided into four major aspects and it will thus be implemented in four phases.

First, we will set up a test bed with a cloud middleware without any modifications. We will verify the test bed and its performance based on benchmarks from the use cases. These benchmarks will provide a baseline for our work.

Second, we will integrate MIKELANGELO components in the test bed manually. In this phase we ensure that the components can run with our cloud deployment. Wherever necessary, changes to components will be made. This phase will mean an initial experimentation with integrating new components, which will then be extended in an automation phase later on. Again, we will run use case implementations to obtain performance data. We will compare this performance data with the baseline measurements obtained in the first case.

Third, we are going to continually modify components and re-run benchmarks. During this project phase we will drive the automation of the whole deployment, testing, and analysis process. Since use case implementations are expected to change, we will ensure that we can flexibly run benchmarks with different setups of components. One important setup will, no doubt, be the original baseline setup.

Finally, we will automate the process of component deployment, testing, and packaging by integrating the whole setup with continuous integration. This work will be based on the incremental improvements achieved in the previous phases.

2.2.2 Research Basis

To provide a basis for feedback on performance and a basis for scientific publications, we need to provide facilities to obtain comparative results.

As input data we will use workloads from our use cases, starting with the big data use case. These benchmarks will be run repeatedly with varying system configurations. To assert our relative performance, we will provide a lower and an upper boundary of performance. The lower performance boundary will be based on running benchmarks on top of a standard cloud deployment. This represents the currently achievable performance and it is what we

work to improve. The upper boundary will be based on a deployment with native or near-native performance. This boundary will represent the best achievable performance, since no or very lightweight virtualisation will be used. To ensure reliable values for comparison, all benchmarks will be run on the exact same hardware.

For each of our MIKELANGELO-enhanced components we will use an equivalent standard component, which is found in widespread use. For example, for sKVM we use KVM for comparison for the lower boundary and Docker for the upper boundary. If possible we will strive to run benchmarks directly in the host system. However, it is foreseeable that this will not always be easily achieved. According to research a Docker-based deployment can achieve results of similar performance as a native deployment, when not writing to the Union File System [2]. In addition, Docker offers far easier management of services than is possible with a native deployment.

In the later project stages, we will automate the process of deploying new components and running benchmarks on top with continuous integration. Based on this work, we will make it possible to get quick feedback on changes to algorithms. Thus, we will be able to improve our results quickly and we will be able to obtain exhaustive data that can be analysed by the research community and that can be used for publications directly.

2.3 Roadmap

The roadmap for the cloud integration consists of three main phases:

1. semi-automated deployment,
2. integration with continuous integration,
3. extension of the cloud middleware.

These phases overlap in part due to iterative improvements of features and due to interrelated effects. The first phase of semi-automated is expected to last until month 18. Then the second phase of integration with CI is expected to take over. Most work in the second phase is expected to be finished by month 24. Nevertheless the second phase will continue with reduced effort until at least month 30. The third phase, extension of the cloud middleware, is expected to last until the end of the project in month 36.

The first phase features semi-automated deployment, which starts out with the manual deployment of a test bed and it ends with a quickly re-deployable system. One of the major goals in this phase is to offer a production-grade test bed. The production-readiness of the deployment is important since we want to do performance tests in a realistic setting.

Furthermore, we plan to integrate the test bed with GWDG's cloud to allow for extended tests and for simplified exploitation. At end of year 1 the first deployment of the test bed shall be ready. This deployment will include snap [3], sKVM [4], OSv [5], and HiBench [6] as an initial benchmark. Even in the initial phases it is important to be able to re-deploy the test system quickly with low overhead, since parts of the whole deployment are expected to change on a daily basis. Furthermore, the partially automated deployment of the system will be an important basis for the automation in the second phase.

The second phase integrates the deployment, verification, and testing of the cloud integration in a continuous integration system. At the core of continuous integration lies the ability to automate the deployment and to provide high-level approaches to run and analyse the results of performance tests. The central elements of this phase are an automation of the cloud middleware's deployment, a middleware for continuous integration, and benchmarking facilities. By the end of Y1 the initial deployment of continuous integration will be made available. However, a broader set of features will be integrated during the remainder of the project. Later in the project, we expect to automate functional and performance tests for separate components as well as for the integrated platform. Performance tests shall be based on use cases as well as relevant benchmarks.

The third phase will extend existing cloud middleware and integrate the MIKELANGELO stack more tightly to take advantage of cross-layer optimisation opportunities. Extensions to the cloud middleware will include features such as live-scheduling based on I/O metrics as objective functions. Further extensions will include deep integration with OSv via the package manager MPM [7], holistic monitoring, and the basis for a big data platform. Cross-layer concerns will optimise for I/O and manage security concerns in the cloud layer, based on work on the hypervisor-level.

This report is written at the end of Y1, after WP5 has been active for six months. Since it is the first report on the cloud integration, the contents revolve mostly around technical details of phase 1.

3 Building the MIKELANGELO Cloud Stack

This section contains a description of all relevant components of the MIKELANGELO stack for cloud computing. The whole stack is conveniently segmented into the host system, the cloud middleware, application management, and cloud monitoring. Each of those segments represents a major part of the work in MIKELANGELO. Each segment poses separate challenges. Thusm we first describe multiple alternatives and reasons for choosing our base technologies over others. Then we describe the work already carried out and finally we briefly sketch out the future steps.

3.1 Hosts, Hypervisor, Guest OS

In this section, we describe the host systems used for the integration with cloud computing. The hosts have been chosen as typical, albeit low-scale cloud hosts, based on GWDG's experience as a cloud provider. The initial setup builds on three compute hosts and a weaker, but separate control node. Once the initial test bed will be fully operational, we will update the node's memory to run larger problems. In later project phases, we will extend the test bed with additional nodes. The current goal is to extend the testbed to six nodes in the first half of year 2. In year 3 we will either extend the testbeds to offer more nodes or extend the memory of the existing nodes.

The testbed consists of six Dell R430 servers [8]. They each have one Intel Xeon processor with eight cores and support sixteen hardware threads at 2.4GHz. The nodes are equipped with four banks of 16GB memory. The motherboard offers eight memory banks, which allows for an extension of memory in future. Thus, in later project stages the nodes may be extended from 64GB to 128GB, which is especially of interest to gauge the performance of big data and HPC workloads. The NICs are standard 10 Gb/s Ethernet interfaces. Each node contains two interfaces configured to run in bonded mode. Thus, the bandwidth of the network can go beyond 10Gb/s and we achieve a higher availability due to redundancy. Network interfaces do not feature RoCE capability. The use of RoCE-capable Infiniband cards is under consideration. However, this would stray from the goal of providing a typical cloud testbed. Currently, we do not foresee that Infiniband will become a commonplace technology in clouds. One of the goals of this testbed in WP5 and T6.2 is the evaluation of big data and cloud workloads in a standard cloud with commodity hardware. A final point should be made on the available storage. The hosts have two 600GB HDDs that are configured in a RAID 1 setup. However, the testbed will use enterprise grade block-storage offered by GWDG's

common infrastructure. Thus, the limited amount of 600GB available storage is of no concern to the use cases.

The hosts run Ubuntu 14.04 LTS, which has been aligned with the HPC testbed at HLRS. Current deployment is based on Linux kernel 3.18-generic with backports from a 4.x kernel. The specific kernel includes patches from IBM with the initial prototype of IOcm running in KVM [9]. For our initial tests we are using Qemu 2.2.0, KVM 3.19 with modifications, and libvirt 1.2.12. The host OS is provisioned by a PXE boot with IPs assigned via DHCP. The provisioning of management components and software happens via Foreman and Puppet as described in Section 3.2.2. The network uses an overlay network using OpenStack Neutron VXLAN as described in the following sections. The hosts will allow VMs to have public IPs, while the host IPs are private. Guest VMs will initially be created using Ubuntu 14.04 LTS and OSv. We will use the most recent development branch of OSv we will use the current version of the master branch.

3.2 Cloud Middleware

In this subsection we describe how we chose our priorities for the integration with a cloud middleware and how we are integrating with this cloud middleware.

3.2.1 Choosing a Cloud Middleware

The choice of a cloud middleware needs to be considered carefully since it would be costly to revert this choice at a later point in time. Our process for choosing a cloud middleware starts by considering relevant products. Based on the initial set of systems we eliminate unsuitable candidates based on technical considerations and for reasons of exploitation.

After nearly a decade since the launch of Amazon Web Services, which popularised cloud computing, and an initial burst of cloud platforms, only a few players are left on the playing field. First of all, we only consider cloud platforms that are completely open source software. Proprietary systems do not lend themselves easily to modifications, make exploitation problematic, and undermine the idea of easy reproducibility of scientific experiments. Currently, there are four major cloud platforms that fit the first criterion of being open source software: OpenNebula, OpenStack, Eucalyptus, and CloudStack.

For those four candidates we have reviewed the feature sets to identify the one or two most promising platforms. A comparison of features is provided in the appendix in a feature matrix for those four platforms. Our technical and exploitation considerations revolve around the features that are striking with regards to our requirements. Thus, these features fuel our decision making process.

The technical considerations revolve broadly around the platform's architecture and its performance. The architectural differences between the four platforms can be found in terms of adaptability, communication methods, and by integration with continuous integration.

The adaptability of platforms is reflected by their supported interfaces and by the platform's extensibility. The interfaces include, for example, supported virtualisation technologies, storage backends, supported databases, and VM image formats. Considerations regarding platform adaptability include the programming language, platform design, and community support.

When considering supported interfaces, Eucalyptus and CloudStack have shortcomings by not supporting Docker as a virtualisation backend. To integrate with sKVM and perform performance tests for comparison, it is helpful to manage an application with KVM and Docker through the same platform. Another downside of Eucalyptus and CloudStack is their lack of support for Ceph, which will likely play a central role as object storage in our test bed in year 2 and year 3. Ceph is becoming an increasingly popular open source alternative to enterprise-grade storage arrays.

Considering the programming languages, OpenStack and CloudStack provide the best extensibility. Both are programmed with only one programming language. OpenStack uses Python, while CloudStack is programmed in Java. Both languages are vastly popular and backed and used by large enterprises. When considering those two languages, python offers a flatter learning curve and overall higher productivity. OpenNebula uses a mixture of Java and Ruby. Requiring two programming languages, one of which is not vastly popular, introduces additional friction when trying to contribute to the platform code. In the case of Eucalyptus the four languages Java, C, Python, and Perl are used for the platform. This introduces significant overhead when trying to contribute code.

Platform design is an important consideration when considering extensibility. Three important aspects of platform design are robustness, scalability, and decoupling of components. In terms of platform design OpenStack offers most benefits over the three other candidates. By using RabbitMQ as a message bus OpenStack allows for a high degree of scalability and for failure robustness. Both come with the additional benefit of being able to distribute various components and to use them in an active-active setup.

Community support is hard to gauge, but can be estimated by the backers of the projects, the users of the platforms, and the activity of open sources repositories and issue tracking systems. Since all projects are available on Github [10, 11, 12, 13] we use GitHub to gauge the activity and community support of those projects. We use the number of commits and the

number of contributors based on GitHub's "Pulse" features. The data is based on the period between 17.11.2015 and 17.12.2015. We are choosing these metrics to gauge the activity of the project and the sustainability, based on the number of contributors. Figure 1 shows the activity for OpenStack Nova, Figure 2 shows the activity for OpenNebula, Figure 3 shows the activity for Eucalyptus, and Figure 4 shows the activity for CloudStack. While OpenStack is split into many sub-projects, the other platforms rely on only one repository to manage their code. We're using these graphs to estimate the general activity of the projects. We should use an aggregate of OpenStack to estimate the real activity. However, we will soon see that Nova alone is far more active than the other platforms as a whole.

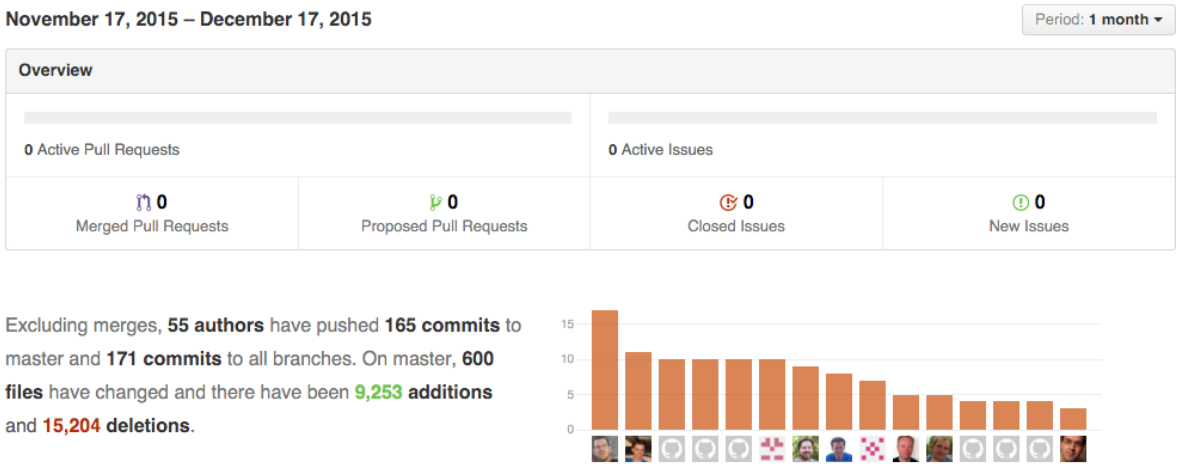

Figure 1: GitHub Activity of OpenStack Nova

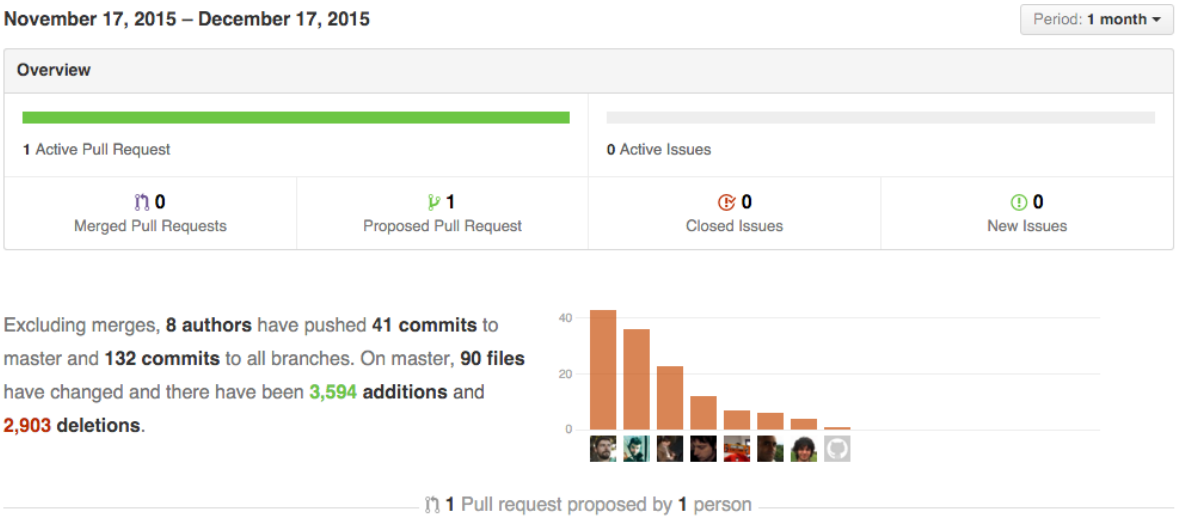

Figure 2: GitHub Activity of OpenNebula

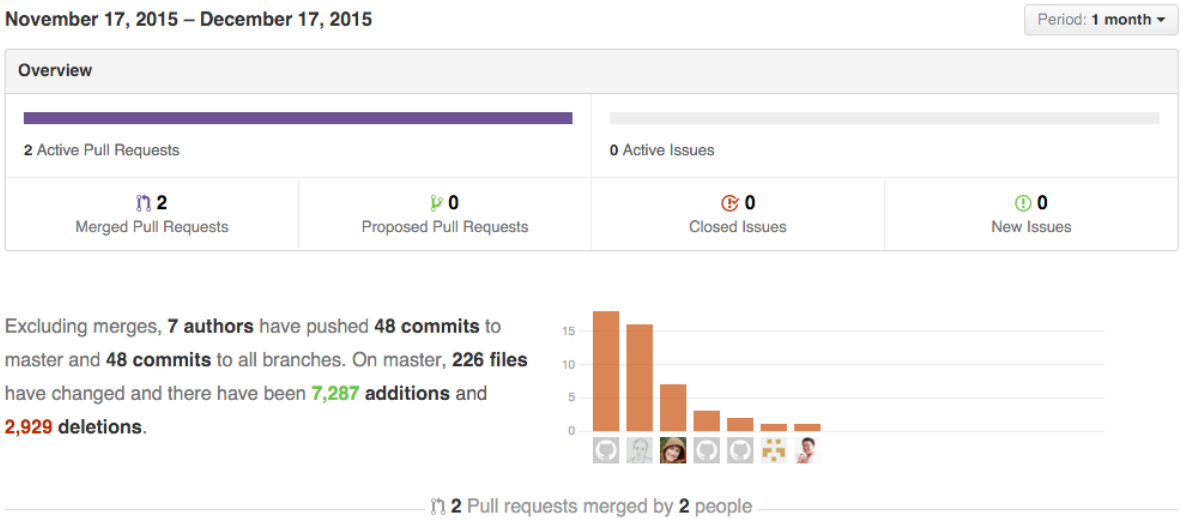

Figure 3: GitHub Activity of Eucalyptus

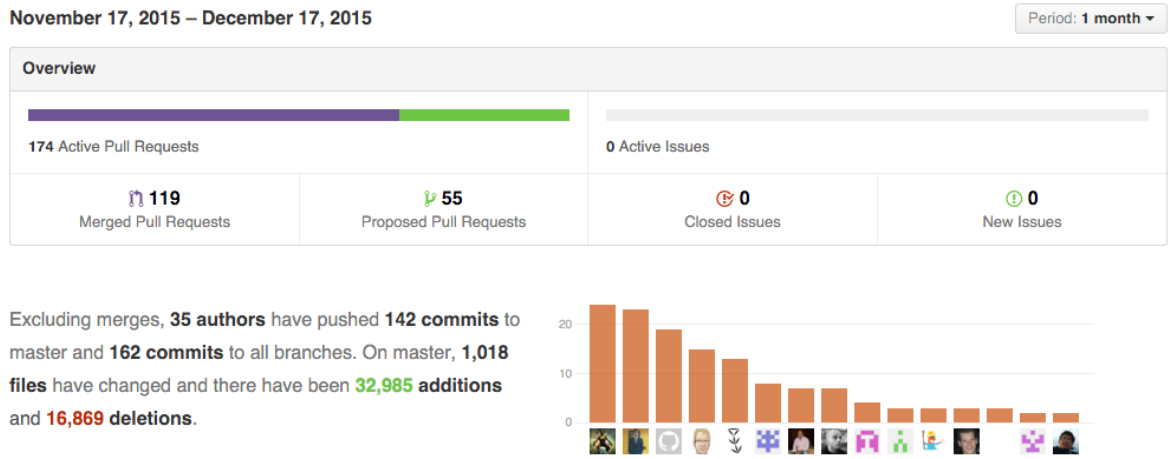

Figure 4: GitHub Activity of CloudStack

In the figures we can see that Nova alone is vastly more active than OpenNebula and Eucalyptus as whole platforms. More authors have contributed more code in more commits during the period shown.CloudStack has more contributions than Nova in the given period, however the number of authors is less. In addition, adding just one additional repository of OpenStack into the calculation let's OpenStack lead again. An important aspect of projects is the activity of developers in the projects. All four contribution graphs show a power-law distribution. However, the distribution is most problematic for OpenNebula and Eucalyptus. There very few developers contribute most of the code. For CloudStack the distribution is more smooth. For OpenStack finally the curve is the smoothest. This metric is important since the withdrawal of individual contributors does not endanger the whole project easily. Thus,

from the viewpoint of the community we have to conclude that OpenStack is most promising, while CloudStack might be a viable alternative.

Communication between components is another important aspect of our technical considerations to choose a cloud platform. The communication method can also be described as the integration approach to the whole platform. The communication methods are important to gauge whether individual components can be modified by themselves. This in turn allows simpler modification during development and a simplified upstreaming. The communication methods further hint at the scalability and failure robustness of the framework. Here CloudStack, Eucalyptus and OpenNebula show a tight integration without emphasis on performance and robustness to the degree that OpenStack offers. OpenStack uses RabbitMQ [32] for communication between components. OpenStack even allows mostly active-active setups based on RabbitMQ for high performance and high availability. The strong emphasis on individual components in OpenStack is reflected by the fact that OpenStack consists of numerous repositories on GitHub, one for each component.

The Integration with a continuous integration system is an important aspect for our choice of a cloud middleware as well. According to the documentation of the projects all offer approaches for continuous integration. In general, it is preferable to be exposed to fewer technology stacks and programming languages. Although OpenStack relies solely on Python, testing OpenStack can be a challenge. The benefits of distributed components listed above become a problem for lightweight testing. OpenStack covers tests for developers by providing a distribution called DevStack [14], which can be used to check for functionality. In our project, however, we will require performance tests, which in turn need to be performed on a full deployment of OpenStack. A full deployment of OpenStack requires a lot of effort for set up. The significant effort required to deploy even a modest OpenStack testbed with a realistic production configuration is the major downside of using OpenStack.

The system performance is the second consideration after the analysis given above for the architectural considerations. Regarding the consideration of performance we only analyse OpenStack and OpenNebula, since in the architectural analysis above it has become evident that CloudStack and Eucalyptus are not a viable approach for us currently. Preliminary analysis liminary analysis of performance is based on publications. These publications show that OpenStack and OpenNebula have comparative performance. However, OpenStack offers slightly better performance when deploying large batches of VMs. In our work the faster deployment of batches of VMs may become relevant in all use cases. HPC scale-out for the bones simulation and OpenFOAM requires an elastic system, which can boot many VMs quickly. The big data case will also rely on batch-deployment to set up fresh virtual big data

clusters. The cloud bursting use case, by definition, relies on a quick deployment of large numbers of VMs.

Exploitation considerations have to be taken into account to choose a cloud middleware, alongside the technical considerations. The exploitation considerations need to include the likelihood for continued development, the future user base of the system, and the ease of upstreaming.

From the viewpoint of backing by business OpenStack offers the best perspective as a large number of enterprises use and develop the framework. This backing is reflected in the contributions of code to OpenStack, which outstrips the other projects by far. OpenStack does not only have the strongest backing, which is reflected by the effort of development shown above. OpenStack also has a diverse amalgamation of contributors [2]. While a few contributors contribute most effort, no organisation in particular dominates the project. Thus, we can conclude that the development of OpenStack for the midterm is ensured.

Gauging the user base of the platforms is hard to do. OpenStack lists [19] many more users than OpenNebula [20]. However, this data is sparse and unreliable. It is hard to come by actual market shares of open source cloud platforms. One way to estimate the amount of users and their importance is to refer to the analysis of the community and code contributions in the paragraphs above. This reasoning relies on the assumption that a larger user base will lead to a more active development and to more contributions. Furthermore, OpenStack is much more active on its mailing lists [21] than OpenNebula was [22] and is now in its forum [23].

Upstreaming is much more problematic for OpenStack than for OpenNebula. This is inherent in the size of the projects. While large projects rely on elaborate processes and tests for contribution, small projects can facilitate short turnaround times and uncomplicated contributions. The difference in effort can be enormous. While upstreaming to smaller projects may take place within weeks, upstreaming to a large project may take up to a year. That introduces considerable friction. However, the elaborate upstreaming has two advantages, as well. First, the elaborate upstreaming process will increase our engagement with the OpenStack community. This increased engagement will fuel our learning process and help exploitation. Second, this elaborate upstreaming process keeps the quality of the software high, despite a wide developer base and a high project complexity.

Taking into account the technical considerations and the implications for exploitation, we conclude that OpenStack is the best way for us to go. Eucalyptus and CloudStack do not

[2] http://stackalytics.com

provide sufficient technical reasons in comparison to OpenNebula and OpenStack. OpenStack offers a range of technical advantages over OpenNebula. These advantages by themselves do not mean that OpenStack should be preferred. However, by also evaluating the aspects of exploitation based on the activity of development, community, and adoption, OpenStack becomes the preferred cloud middleware for integration in this project.

3.2.2 Integration

In this section we describe the integration of OpenStack with MIKELANGELO. During the first months this integration mostly refers to considerations on how to run an OpenStack testbed to test MIKELANGELO components. This testbed will need to serve for performance tests, use case tests, and continuous integration.

Our integration with OpenStack is a project in itself. We briefly describe our motivation for this sub-project, our setup, and then the resulting features. Most of the work concerns the integration with a production-grade OpenStack installation. However, we also use an installation of DevStack. The DevStack installation is trivial in comparison to the full test bed. Unfortunately, the DevStack installation is only useful for performing certain types of functional tests. Thus, we will use DevStack to cover functional tests and the full testbed to cover functional and performance tests.

The motivations to integrate with a fully functional OpenStack installation are manifold. First, we want to ensure that MIKELANGELO works in a realistic scenario. Thus, we require a full installation of OpenStack, which differs clearly from a DevStack installation. Second, our full testbed will offer the possibility to use multiple backend systems such as Ceph and storage area network (SAN) appliances for block storage and object storage. Also the networking configurations can only be tested realistically with a full OpenStack deployment. Third, a central aspect is the capability to perform performance tests easily. One of the major goals of MIKELANGELO is a clear improvement of I/O performance. To test this, we require a system that will perform as a production system will. Finally, we want to provide deployment scripts for OpenStack so that it becomes easy to use the MIKELANGELO stack for cloud computing for private and public clouds.

The setup of our testbed is shown in Figure 5 and Figure 6. Figure 5 shows the general setup of using host machines with Ubuntu running KVM, and in our case also sKVM. In this diagram we are leaving out alternative virtualisation and management technologies such as Docker and Kubernetes. As cloud middleware we are adding OpenStack. The deployment of the system is performed using Foreman and Puppet.

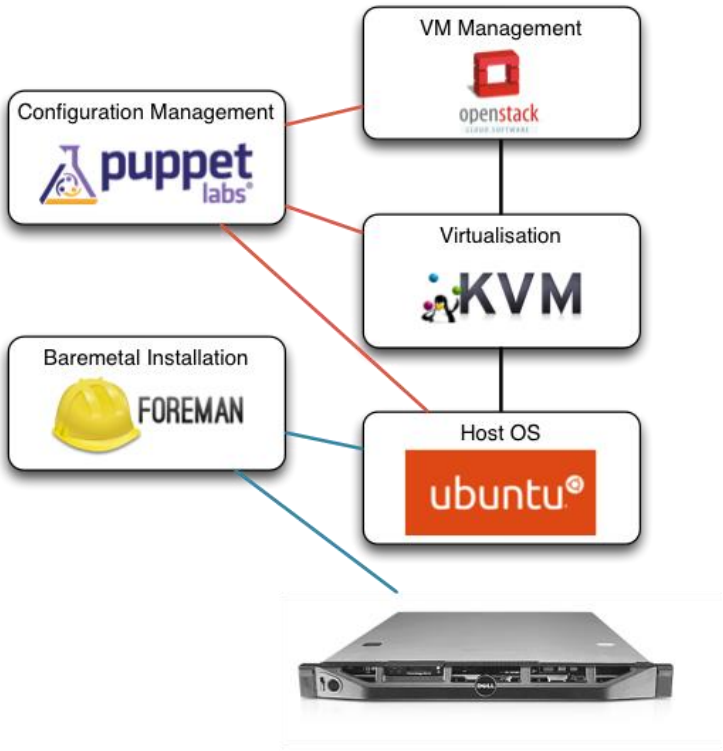

Figure 5: Architecture diagram of the cloud test bed.

Figure 6 shows a more detailed view on our concrete testbed based on the OpenStack architecture. The architecture consists of control and compute nodes.

Control nodes run the OpenStack components that make up the management layer of OpenStack. These include image management via Glance, network management via Quantum, scheduling via Nova, and monitoring via snap. Our setup runs each of the control node services in Linux containers. Most of the services are configured for a redundant deployment in an active-active setup. In the case where multiple control nodes are deployed, the HAProxy allows for the checking of the availability of the other control node, as well as load balancing. The control nodes in general run on less powerful nodes with less memory as their workload is mostly CPU-bound. In our initial testbed there will only be one control node, deployed on one of the host systems. The final version of the testbed will include two control nodes as shown in the diagram.

The compute nodes host the virtual machines. Thus, the hosts for compute nodes usually offer large amounts of memory in addition to powerful CPUs. Our setup supports KVM and sKVM as the underlying hypervisor. In addition, OpenStack allows the definition of host aggregates, which allow to use different virtualisation technologies on hosts. Currently, the deployment of Docker containers is interesting and will be offered in our testbed. Docker

offers a great way to compare the performance of sKVM to near-native performance of Docker. At the same time Docker offers encapsulation of services and simple management via OpenStack. In addition to Docker, there is development going on to use the upcoming Kubernetes within OpenStack. We will evaluate the use of Kubernetes for comparison in future iterations. For network management we use OpenStack Neutron DVR network. As storage backend for VM images we use GWDG's production storage systems. The two configured systems are a Ceph-cluster and an EMC Isilon appliance.

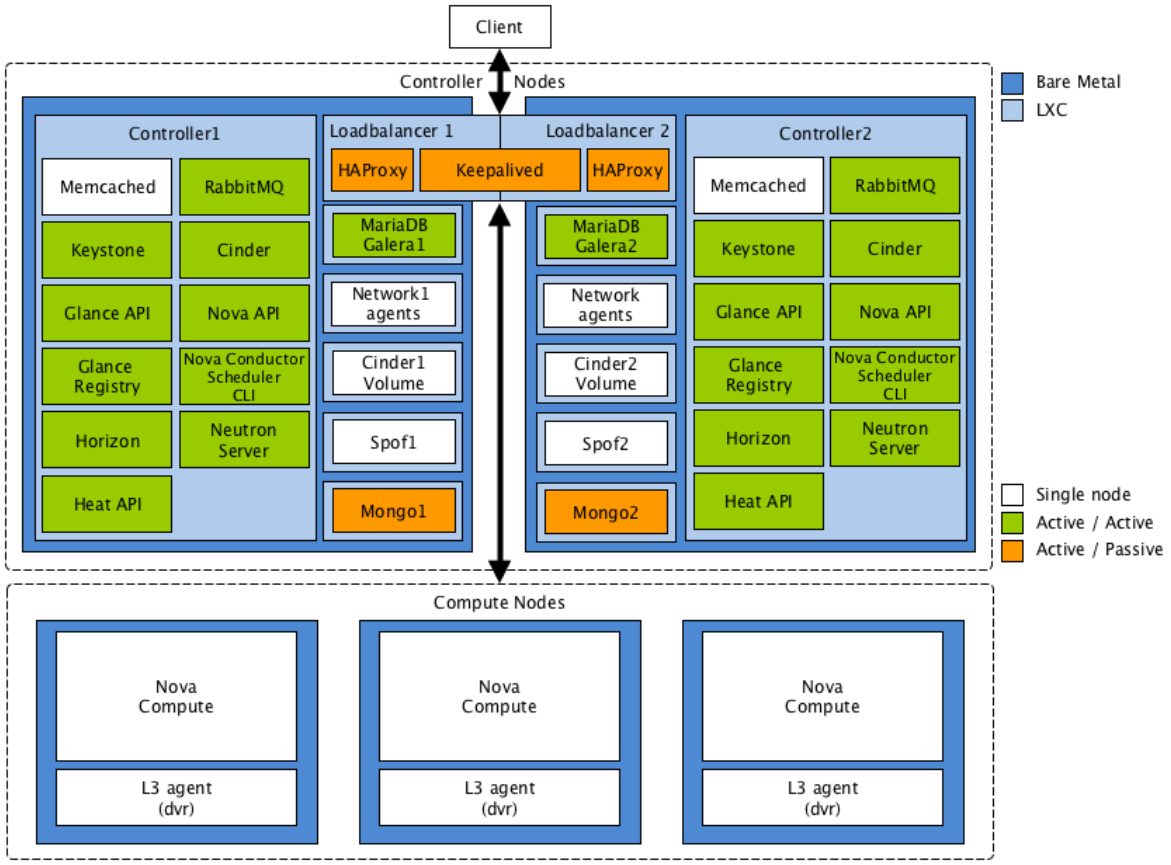

Figure 6: OpenStack-based setup of the cloud test bed.

This setup is similar to the production deployment of OpenStack at GWDG. There are two major differences to GWDG's cloud. First, the testbed runs on far less nodes, which individually are also less powerful. Second, the GWDG cloud is laid out for high-availability, which the MIKELANGELO cloud testbed is not initially. This alignment has two major benefits. First, it becomes possible to use GWDG's cloud hosts to run larger experiments than possible on the MIKELANGELO testbed. Second, the parallel development going on for both systems allows for synergies and for easy exploitation. Thus, GWDG will be in the position to apply results from MIKELANGELO more easily.

The deployment of the testbed is performed using Foreman and Puppet, which leads to multiple advantages. First, a similar setup can be reproduced in other data centres quite easily. This reduces the barrier to adoption for external parties. Second, a mostly automated deployment allows for simple internal exploitation by partners, especially GWDG. Third, we will be able to roll out new versions of the test bed based on internal development as well as external development such as new versions of OpenStack quickly. Finally, we can use the automation as part of elaborate integration tests in continuous integration. New versions of components, including low-level components, such as sKVM, can be deployed and tested automatically, running big data and HPC workloads.

The described testbed and cloud integration allows to harness the full performance of the MIKELANGELO stack. This performance is important to evaluate our results in a realistic scenario for cloud computing. This evaluation will be further facilitated by allowing the use of technologies such as Docker for comparison. Finally, the described setup can be used as part of continuous integration and for frictionless adoption by third parties.

The described integration is currently in development. During later phases of the project this integration will be extended to include a deeper integration of OpenStack with sKVM, OSv, and snap. We are especially excited about the potential to perform cross-layer optimisation, combining sKVM, OSv, snap, and OpenStack's live migration facilities.

3.3 Application Management

Application management is at the core of every dynamic enterprise ecosystem. It is responsible for managing the entire lifecycle of an application, starting with the appropriate packaging of all the required components. Once deployed, it is concerned with maintenance, versioning and upgrading of underlying applications and components.

Package management systems, such as RPM and Debian's APT have been around for nearly two decades. Over the years they have evolved into complex systems dealing with numerous scenarios. They have become de facto standards for package distribution and management that enterprise grade systems support out of the box. This greatly affects the general interoperability of applications. It furthermore separates the application and its dependencies from the underlying system. However, these systems are extremely complex to comprehend and master in entirety. Their learning curve is steep both for application maintainer and for the end user navigating the rich landscape of application packages.

Within MIKELANGELO application management we are primarily concerned with building a streamlined application management system for OSv. OSv is a lightweight operating system

specifically designed for the vast virtual environments offered by today's cloud. Although OSv has been built to be POSIX compliant, some design decisions limit the exploitation of existing application packages suitable for other systems, such as Linux.

To this end, the makers of OSv offer two ways to package and run applications on top of OSv: build scripts and Capstan. The former requires a development environment and the use of OSv source code, which makes it impractical for wider adoption. On the other hand, Capstan introduced a Docker-like system for packaging and running applications but with a major drawback: it treats applications as virtual machine images. This limitation prevents reuse of packages that are used to compose the high level application.

MIKELANGELO's vision is to extend these two approaches and bring some of the best practices of existing systems into the OSv world. MIKELANGELO will significantly simplify the way applications are composed, shared and distributed. Application maintainers are going to be able to use packages as building blocks for composing complex solutions. On the other hand, business stakeholders will benefit from rich customisation options of pre-built packages offering more control over application content to business stakeholders.

However, in recent years the complexity of applications and, in particular, high performing and reliable configurations has grown significantly. Albeit still an important part of an application lifecycle, end users are no longer interested only in deploying and managing standalone applications. End users build and consume entire application stacks, comprised of database management systems, application servers, message passing buses and high availability components, to name just a few.

In the next section we will briefly introduce various OpenStack services from the perspective of application management. We are then going to build on this to present a seamless integration of the MIKELANGELO application management system into OpenStack.

3.3.1 Leveraging OpenStack for Application Management

When it comes to management of application lifecycle, several existing OpenStack services may be considered: Glance, Heat, Sahara, and Murano.

OpenStack Glance

Glance provides a service for storing data resources that are used by other OpenStack services, such as virtual machine images (VMI). These images can range from simple ones containing nothing more than a pre-installed guest OS to images comprised of several applications pre-configured to offer a turn-key solution. Because of this flexibility Glance can be used by application maintainers and cloud administrators facilitating the distribution of

required applications and services. However, Glance is a low-level service: it literary stores complete raw images. When a change in configuration is required, it has to be made on a running virtual instance or by mounting the image file.

OpenStack Heat

Heat facilitates orchestration of multiple cloud applications. It uses OpenStack core services such as Nova, Glance and Neutron to deploy and configure the landscape required by the target application stack. Even though it is primarily focusing on the management of the underlying infrastructure, it can also be used to configure the software components and establish relationships between these components (for example, create a link between a database and the application or between a load balancer and a backend service).

Heat uses Glance to retrieve images when deploying on the managed infrastructure.

OpenStack Sahara

Sahara has been designed to simplify deployment of data-intensive applications. It is primarily a mashing service that integrates various other OpenStack services into an analytical platform. It is not a general purpose application management solution because it uses predefined applications, i.e. the application stack is not customisable. Nevertheless, it demonstrates the importance of various components working together to achieve certain goals. Within MIKELANGELO, Sahara may be used for the big data use case.

OpenStack Murano

Murano is the application catalogue for OpenStack providing cloud users a browsable repository of composite ready-to-use application stacks. These applications may be configured and launched through a well-designed and customisable user interface. Murano specialises in managing the entire lifecycle of applications. It provides application developers and maintainers a high level API for describing the application and its configuration options as well as specifying the process of application deployment. Underneath, it uses other OpenStack services that manage infrastructure and monitoring.

3.3.2 Integrating MPM with OpenStack

In report D4.7 "First version of the application packages" [7] we have introduced the current status of the preliminary version of the MIKELANGELO Package manager (MPM). MPM is currently based on Capstan [24], which is solely a CLI (Command Line Interface) tool for building and running virtual machine images. This makes it extremely impractical to integrate with other systems, such as OpenStack for Cloud and Torque for HPC. To this end, the report

D4.7 also outlined some of the future plans related to improving the integration capabilities of MPM.

MPM with OpenStack will be integrated in both directions:

- use of OpenStack services by MPM
- implementation of specific OpenStack drivers for accessing MPM

MPM currently implements its own internal repository of application packages, images and instances. This is ideal for individual users or small teams, but maintaining another separate repository of cloud-ready applications in a cloud environment becomes tedious and error-prone. To this end, we are going to integrate the API of the Glance service into MPM to have a two-way access to a shared store of application packages and images. Glance presumes that the resources stored in a backend storage are virtual machines that can be used by Nova. MPM, on the other hand, works with application packages that can be reused and composed into a single application virtual machine image. Because packages only make sense when they are deployed in a VM, we are going to extend the (meta-) data model of Glance to support storage of packages not meant to be directly launched. This is in fact in line with Glance's vision of being a central place for application-specific resources.

We are also considering basic integration with Nova giving MPM users the ability to launch virtual machines using MPM commands. Integration with Nova would speed up the development of application packages because testing of these packages would be completely integrated into the workflow. Such an integration would furthermore completely relieve the user from having to setup a local development environment. Nevertheless, this integration is not meant to replace standard Nova clients (for example CLI or OpenStack Dashboard).

Using the functionalities of MPM from within OpenStack services is the other side of the story that we plan to pursue. Murano offers a great deal of flexibility that application maintainers benefit from, but in order to be fully supported a collaboration between the Murano engine (running on controller nodes) and Murano agents (running within deployed virtual machines) is required. Murano agents do the deployment of the actual applications inside VMs.

In order to allow seamless composition of applications from ready-to-use application packages we are going to extend the mechanism of deploying custom applications using MPM. MPM will serve as an agent communicating with the engine itself to support deployment and configuration of required applications.

3.3.3 Deploying OpenFOAM with MPM and OpenStack

To demonstrate how we envision the use of OpenStack and the MIKELANGELO Package Manager, this section presents a scenario of one of the use cases MIKELANGELO will be evaluated with. The Aerodynamics use case [16] is interested in quickly simulating one part of an aircraft or even the entire aircraft under many different physical conditions. To do this, numerous small-scale simulations are required to assess the behaviour, but the current workflow results in significant overhead of configuring input parameters, scheduling and running simulations, and collecting results. Details are out of scope of this deliverable, but can be reviewed in report D2.10 *The First Aerodynamic Map Use Case Implementation Strategy* [16] which describes the purpose of this use case and the business value thereof.

The aforementioned workflow may be simplified to a great extent using MPM and OpenStack. First, using MPM a set of application specific packages will be created. This will include OpenFOAM core libraries, various OpenFOAM solvers doing the actual simulation, the OpenMPI package supporting parallel execution and supporting OpenFOAM tools. All these packages will be stored in Glance as application packages, instead of virtual machine images.

The end user, an aircraft designer without any mandatory IT background (apart from understanding OpenFOAM which is an expert system) will then be able to compose one or more simulations using Murano. She will select the OpenFOAM simulation solver and provide the input data, possibly through the OpenStack object storage. She will also be given a chance to prescribe how each individual simulation should alter the parameters of the input data, all using the central user interface provided by OpenStack Horizon and Murano. Finally, after all the simulations are finished, she will be able to retrieve all the data, final and intermediate that will help in determining the quality of the design.

Although the scenario is yet to be fully defined, it already outlines the main benefits of integrating complex applications into the cloud management layer for the end users. Building on an existing application catalogue (Murano) and integrating it with MPM will simplify the use of lightweight OSv-based applications.

3.4 Monitoring

A complete instrumentation and monitoring stack is being constructed in Task 5.3 to meet the specific needs of the MIKELANGELO project. This work is creating a scalable, highly extensible, telemetry gathering and processing stack, the progress of which can be seen in

Deliverable D5.7 [3], First Report on the Instrumentation and Monitoring of the complete MIKELANGELO software stack.

The design and construction of the Cloud testbed has progressed in parallel and in close consultation with the instrumentation and monitoring efforts. Requirements were fed into the scoping of the work, and are being clarified, prioritised and managed via regular conference calls.

At the time of writing, the instrumentation and monitoring framework, built on the recently open-sourced snap telemetry framework, now includes the functionality to

- capture hundreds of metrics from the MIKELANGELO cloud hardware, host operating system, hypervisor (sKVM), Docker infrastructure and any hosted instances of OSv.
- calculate moving averages of data as required
- publish data to InfluxDB
- expose data via customisable Grafana dashboards

Additionally, snap supports

- arbitrary additional functionality via an extensive plugin-based architecture
- signing of plugins
- encryption of telemetry data transmission
- tribe-based distributed management
- remote, dynamic configuration through command line and RESTful interfaces

As the cloud testbed infrastructure construction and configuration completes, snap will be deployed on all hosts and plugins to capture as much data as is possible from the complete hardware and software stack. Appropriate metrics will be routed through the Moving Average plugin, and the Anomaly Detection plugin when it becomes available. This latter plugin will avoid unnecessary transmission of static telemetry data: high resolution data will only be transmitted when anomalies are detected.

The initial intent is for cloud testbed telemetry data to be fed into an InfluxDB database instance, from which Grafana based dashboards and other tools will be used to query, investigate and compare results of experimental runs. As monitoring needs are further refined, inputs will be fed into the Instrumentation and Monitoring activities to help develop as useful a Performance Analysis platform as possible.

4 Ensuring Quality with Continuous Integration

Continuous integration is used to ensure a quality output of MIKELANGELO in a cloud setting. Continuous integration is a widely applied technique in software engineering nowadays. Nevertheless, even in relatively small projects, continuous integration may pose significant challenges in implementation. The challenges in MIKELANGELO go beyond those of enterprise-like applications. We need to test the whole computing stack automatically and quickly.

The challenge of ensuring quality in MIKELANGELO lies in our whole-stack approach from the hypervisor, which is embedded in the host OS kernel, to high-level algorithms running in virtualised clusters managed by OpenStack. It is a challenge to integrate those components smoothly altogether, since it takes some time to set up and run. However, for smooth development, we need to create such a setup, run experiments, and provide insights to the results quickly. Thus, our only hope is automation of this process.

Continuous integration allows us to automate the whole setup of the MIKELANGELO stack. There will be different queues for testing, to allow scaling of the testing procedure. For example, one queue will contain tests that simply build a kernel and test for functionality and performance. At the other extreme, there will be queues that will assemble a specific version of the whole MIKELANGELO cloud stack and test it with big data workloads. Automating the testing of the variety of components entails the automation of the whole provisioning process. Here the deployment scripts for OpenStack will be put to use.

The work on continuous integration is in its early stages. Currently, Jenkins [30], a software for continuous integration, and Zuul [31], a software for testing OpenStack, are set up. The system now will need to be filled with tests for various components and integration scenarios. Thus, during the project's lifetime, the continuous integration system will grow. An intermediate goal is to have a testing procedure at the end of the second project year that will allow upstreaming to various projects.

Continuous integration is a large endeavour that spans the whole stack and all use cases. Thus, the work is carried out across work packages. The initial description is provided here. Further work on continuous integration will take place in other work packages. Especially, in work package 6 the implementation of individual tests will be undertaken.

In Section 4.1 we first describe our requirements for deployment, testing, and packaging. In Section 4.2 we then present the current concept of how to implement continuous integration for MIKELANGELO.

4.1 Requirements for Deployment, Testing, and Packaging

This section discusses the requirements for deployment, testing, and packaging of MIKELANGELO components for cloud computing. Without a continuous integration system these components would need to be integrated, tested, and packaged manually.

The deployment of components refers to the installation and configuration of components in an integrated environment. Testing refers to functional and performance tests. These tests comprise isolated tests and integrated tests. Isolated tests just test a component with any means, such as benchmarks using mocked components in the periphery. Integrated tests test one or more components that are connected. These tests aim to verify that those components work well when plugged together.

The following sections discuss the basic considerations for our main components: sKVM, OSv, OpenStack, and snap.

4.1.1 Deployment, Testing, and Packaging of sKVM

Our hypervisor, sKVM, contains multiple modifications and new components. Since they share commonalities for deployment, testing, and packaging, we summarise all three under sKVM. The main innovations in sKVM are IOcm [4], vRDMA [4], and SCAM [4]. All three components are part of sKVM and thus part of the Linux kernel. The project has agreed to use the 3.18 Linux kernel as a basis.

Building these components means to build the commonly patched 3.18 kernel. Deployment works by booting into this kernel. For the initial project phases this kernel will run as part of Ubuntu 14.04 LTS.

Tests for those components have not yet been agreed. However, it is foreseeable that functional tests will require the creation of VMs. Likely those tests will be implemented by using Bash and Python scripts, and C code. Performance tests for all three components will use specific benchmarking tools.

In an integrated scenario, sKVM will be the major endpoint in most tests. In a production setup of the MIKELANGELO cloud stack all other components will rely on sKVM. Thus, the integrated testing scenarios for OpenStack and cloud-based use cases will test sKVM thoroughly for functionality and performance.

4.1.2 Deployment, Testing, and Packaging of OSv

OSv is our preferred guest OS for the cloud. Our work on OSv includes mostly the extension of OSv features to run use case applications and to improve their performance.

Building and deployment of OSv works with OSv's own scripts. These scripts need to be called with provided application binaries. These binaries need to be built upfront on a Linux system. The build process often requires some modification to produce position independent code in a shared object. The built binaries can either be provided to a running instance of OSv or they can be built into a new OSv image.

The initial deployment of applications within OSv needs to be performed manually. This process requires insight from developers and modification of the build process. However, once the process of deploying and running an application in OSv has worked out the process can be automated. This automation can indicate whether an application runs in OSv. Additional tests for functionality will include the functionality of application deployment with MPM, described in Section 3.3. These tests will be performed in conjunction with OpenStack. The automation of testing can also provide performance indicators. Performance tests will rely on specific applications and their respective benchmarks. Currently, performance tests for NoSQL databases [15], OpenFOAM [16], and big data are in use [17].

In an integrated scenario OSv will depend on sKVM only. The cloud components and the big data components, however, will depend on OSv. Again, the use case implementations will test OSv and its interplay with sKVM thoroughly.

4.1.3 Deployment, Testing, and Packaging of OpenStack

OpenStack is our preferred middleware for cloud computing and will receive various contributions. Thus, OpenStack needs to be tested thoroughly as well. The resulting versions of OpenStack need also to be deployed upfront for testing. Successfully tested versions need to be packaged for installation by third parties.

Testing of OpenStack is complicated as its deployment in a productive environment is complicated. For functionality tests we can rely on DevStack, which makes matters simpler. However, we also aim for improved performance. Thus, our tests need to incorporate performance tests, which in turn require a production deployment. These considerations are described in detail in Section 4.2.

Fortunately, a couple of tools exist to ease the process of testing OpenStack. We mainly use the four tools DevStack, Zuul, Rally, and Puppet.

DevStack allows us to develop parts of OpenStack without a full deployment. This tool is especially helpful for developers. In the context of continuous integration we will rely on DevStack to perform functional tests of the OpenStack components. Thus, DevStack allows

for functional tests in a partially integrated scenario. The functionality of the OpenStack components can be fully tested. Furthermore, DevStack is not suited for performance tests.

Zuul is a testing environment for OpenStack that integrates with Jenkins. Zuul is the tool used for testing by the OpenStack community. We will rely heavily on Zuul to integrate with Jenkins, and perform most of our tests that rely on the full cloud stack. The use of Zuul will furthermore facilitate the upstreaming to OpenStack. We describe the use of Zuul in more detail in Section 4.2 below.

Rally is a performance analysis tool for OpenStack. The performance tests of our cloud integration that are directly related to OpenStack will be performed using Rally. We will augment these measurements by specific measurements for other components, however.

Puppet is being used to deploy the production-level test bed. Our self-developed Puppet scripts allow the deployment of a full OpenStack system. These scripts allow us to test components in an integrated environment on a regular basis. These scripts will be called in Jenkins jobs to set up the fully integrated test environment.

Finally, packaging of OpenStack happens by following the official guidelines [18]. This process is time consuming but easy to automate. Thus, we will integrate it with Jenkins. This integration will allow us to release stable versions of the MIKELANGELO cloud stack on a regular basis.

4.1.4 Deployment, Testing, and Packaging of Snap

Snap is a new cloud monitoring framework by Intel [25]. Since snap is such a new framework, bugs and missing features will likely arise. Thus, thorough testing will become even more important for this component

The goal of snap is to provide a holistic and adaptable framework for monitoring. The integration of snap into MIKELANGELO is described in Section 4.2. The system itself is written in Go. Building of the components is straightforward thanks to the use of git and make. However, testing of the integrated framework poses a challenge.

First, after building the framework needs to be deployed automatically on our test bed. This deployment includes an exemplary configuration. Then functional and performance tests can be performed on the test bed. Performance tests can be carried out by increasing the resolution of monitoring. Functional tests however will need to focus on communication between performance and on errors thrown by the framework. Full functionality testing of snap would require statistics and a redundant system to verify that the recorded data is

correct. Furthermore, in the context of MIKELANGELO, only snap plugins developed in the project will be tested as part of the cloud integration.

Packaging of snap is already covered by its own development and deployment process, described on its github page. In MIKELANGELO we are going to integrate this process with Jenkins to provide ready-to-use packages in regular intervals.

4.2 An Architecture for Deployment, Testing, and Packaging

This section describes our approach to perform the tasks described in the previous section in an automated way.

To provide timely feedback to developers and to cope with the complexity of our overall system, we are using a combination of tools. The pivotal component of our cloud setup is OpenStack since its deployment is most complex and it serves as a bridge between the virtual infrastructure on one side and the use case implementations on the other. Thus, our central component for testing and automating this pivotal component is Zuul. Zuul has been described in the previous section already.

Figure 7 shows the current architecture for continuous integration for cloud computing. The overall system is set up mostly at GWDG. GWDG provides the testbed as described in Section 3.1. The figure below concentrates on the continuous integration architecture, which in turn integrates with the cloud testbed architecture. The figure shows on the right-hand side Zuul and Jenkins. Both components are deployed using Docker. Zuul is furthermore integrated with Gerrit. Gerrit is a tool for code review, which supports Gitlab, a tool for git hosting. Gerrit and Gitlab are hosted by XLAB.

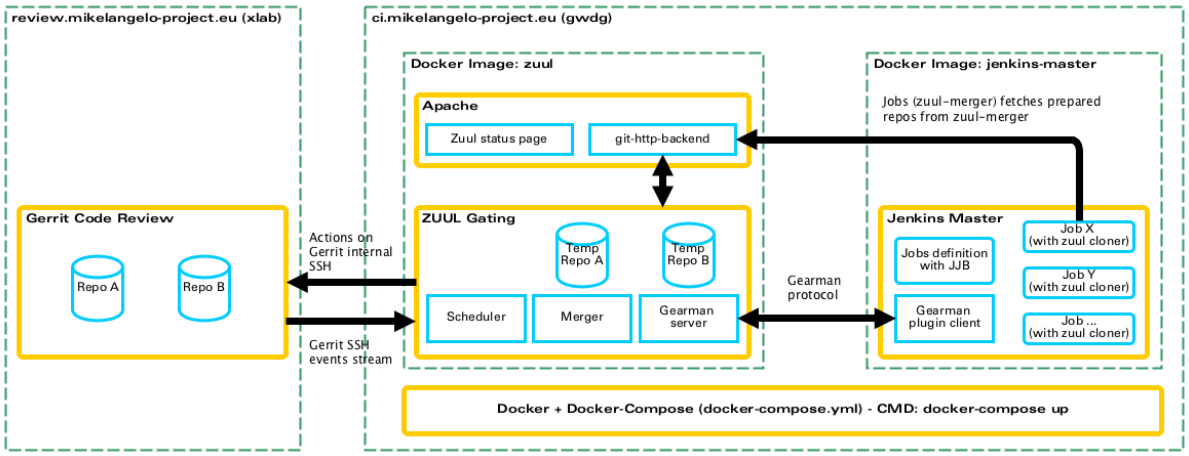

Figure 7: The architecture for continuous integration of the cloud integration.

The basic process for testing is triggered by a push to certain branches in Gitlab. Pushes to stable branches trigger component tests for functionality and performance once they are approved in Gerrit. These changes are monitored by Zuul, which then sorts these changes to one of the testing queues. Some of the testing queues are for long-running tests while others are for short-term tests. Long-running tests may take several hours to perform and they will block some of the hosts in the testbed. .

Tests are triggered in Jenkins, which contains job descriptions for those tests. These job descriptions are responsible for running tests and for collecting the resulting data. The individual tests will be provided by the component and test-owners. The range of tests spans from functional component tests to functional and performance tests in integration. The latter may include running use case code on a virtual cluster with experimental components throughout the system. Such a test may even run for multiple days.

As of writing this report, the continuous integration system is under development. The basic setup is available and needs to be extended to allow for first tests by partners. Availability of the continuous integration setup is expected in January 2016. The system will initially be verified by running the build process for our common kernel. This test will thus cover basic tests for sKVM, including IOcm, vRDMA, and SCAM.

Future iterations of this report will document the progress of the continuous integration platform and the available tests, their coverage, and the test results.

5 Implementing a First Demonstrator

This section summarises the work on the Y1 cloud demonstrator. This demonstrator is aimed as a deliverable in M14 at the first project review. As of writing this report the work on the demonstrator is ongoing. Thus, we describe the expected feature set, without giving guarantees for their availability. Unforeseen complications can preempt our efforts to provide the targeted feature set.

The cloud demonstrator for year 1 combines most of the MIKELANGELO cloud stack with a big data workload. Thus, the demonstrator will show most of the cloud integration described in Section 3 in a very early stage of development. Most components are expected to be available in the demonstrator. However, these components will be in an early stage of development and rather of prototypical character. The following sections describe the experiments run in our demonstrator, the integrated test bed, and different experiments to be run with a big data benchmarking suite.

5.1 Experimental Design

The experimental design for the demonstrator is influenced by three components: the big data benchmark, the cloud integration, and our components to be showcased.

As big data benchmark we are using HiBench [6], which will be described in some more detail in Section 5.3. HiBench will be used as the application to run in our demonstrator. It provides the workload and measures basic performance metrics, such as throughput and latencies of jobs. To obtain data for comparison HiBench will be run using three major configurations: a traditional stack, the MIKELANGELO stack, and a containerised stack. The traditional stack uses KVM, an Ubuntu guest, and standard virtio to run the benchmark. The MIKELANGELO stack uses various combinations of MIKELANGELO's components. These configurations are described below in Section 5.2. The containerised stack will use Docker for the initial setup. We expect that Docker will provide an upper boundary for the performance while the traditional virtualisation stack will provide a lower boundary. The MIKELANGELO stack is expected to be slightly above the lower boundary for the initial demonstrator. The project's goal is to move this performance as far up as possible.

Since HiBench is of such importance to us, we describe the basic setup of HiBench. HiBench is an open-source benchmark for big data applications and is developed by Intel. The benchmark uses a set of ten microbenchmarks that represent typical big data workloads. These benchmarks are implemented with different technologies such as Hadoop, Spark, and Storm using different languages such as Java, Scala, and Python. Thus, HiBench relies on the

availability of a full big data cluster. In our case, this cluster will be made available in Docker, a Ubuntu guest, and, if possible, OSv.

When invoked, HiBench runs all ten microworkloads based on pre-configured workloads. We will repeat these experiments with different system configurations. The resulting performance data will be analysed, to obtain insights into the differences in performance between virtualised environments and native ones. The benchmark works by first generating data sets and then running corresponding big data algorithms such as sorting and searching algorithms. The major results are performance recordings. Aggregate performance records show the throughput of the whole cluster and the execution times for all microworkloads. More detailed data shows the throughput of individual nodes for individual microworkloads. Thus, a lot of data is generated that allows for analysis of bottlenecks. To augment the results, we will work to leverage snap's features to process additional performance monitoring data collected from individual hosts and VMs.

The next section describes more concretely the setup of infrastructure components that will run HiBench and collect resulting data.

5.2 Test Bed

The testbed for our demonstrator consists of various components and will analyse their relative performance for the HiBench benchmark.

Since MIKELANGELO covers a multitude of components, ideally, we would like to analyse the impact of each in detail. Then we would like to analyse the interplay of those components to perform cross-level optimisation. However, for an initial demonstrator this is too much work to carry out with the given effort. We will perform such experiments and analysis later in the project. These experiments will build on the work on continuous integration, which first has to progress further. For the initial demonstrator we aim for three deployments. The first stack will feature a traditional cloud stack with no MIKELANGELO-based components. The second stack will feature a Docker-based stack, again without any MIKELANGELO-based components. The third stack will include as many MIKELANGELO components as can possibly be demonstrated at the end of the first project year.

The lack of a native deployment, that is without any virtualisation, in these configurations is obvious. The main reason for this lack is the difficulty to automate the native deployment of HiBench and the integration with continuous integration. Although this is possible by itself, the automation becomes much harder and more cumbersome when considering that the hosts also run OpenStack and Docker. This problem becomes even more significant once we

start including use case implementations in our tests. Thus, using Docker as a surrogate for a native deployment allows for simple management, which makes an automated integration with continuous integration feasible. Furthermore, Docker has only a small performance impact when compared to native performance. The main performance hit when using Docker arises from writing to the union file system [2], which we will avoid.

In the following subsections we review the virtualisation technologies, guest OSs, vRDMA, and monitoring components for our experiments.

5.2.1 Virtualisation: KVM, sKVM, and Docker

Regarding virtualisation technology the experiments will build on KVM, sKVM, and Docker to establish upper and lower boundaries for the performance.

Although we name sKVM and KVM separately, actually both are the same system. Our hypervisor, sKVM, is a modified version of KVM. Since this modification to KVM is additive, it is possible to configure sKVM to be equivalent to KVM. In our testbed we will thus deploy sKVM only. According to our needs, we will configure sKVM to behave like KVM or to provide extra I/O performance. The main difference between sKVM and KVM is thus the activity of IOcm, vRDMA, and SCAM. Each of those components can be deactivated. In the case of IOcm this configuration means that no I/O cores are deployed. This approach is very convenient when compared to the alternative of re-deploying a different kernel.

As an alternative virtualisation technology, we use Docker. Docker uses Linux containers for virtualisation, which provide near-native performance. A significant performance loss can be observed in Docker when using the internal file system, called union filesystem, excessively. Being aware of this caveat, we will deploy our experiments to use the host's native filesystem.

In our demonstrator, we will compare all three types of virtualisation: KVM, sKVM, and Docker. We expect KVM to perform the worst. With sKVM we will use a configuration of IOcm that will provide the best-possible performance. Docker finally is expected to outperform both KVM and sKVM in terms of I/O throughput and latency.

5.2.2 Guest OS: OSv and Ubuntu

As guest OS the demonstrator will use Ubuntu and OSv.

Ubuntu is chosen as a widely-used cloud OS. At the same time the typical Ubuntu guest runs a nearly full Linux distribution, which comes with a lot of legacy and hardware support that acts as unnecessary baggage in a typical cloud deployment. OSv in turn is a new operating system that is specifically built to be run in the cloud. Ubuntu will be used as part of the

traditional cloud stack, while OSv will be used as part of the MIKELANGELO cloud stack. Although we could allow for permutations of these configurations, they do not serve our purpose. The main purpose of this demonstrator is to showcase the total speedup gained by using MIKELANGELO when compared to a traditional virtualisation stack. Docker, finally, does not require a guest OS, since it uses the host's kernel.

The goal of using OSv is to show that some components can actually be run using OSv. So using OSv is rather a matter of demonstrating functionality. For the demonstrator we are working to run as many big data components on which HiBench depends in OSv as we can muster. Initially the demonstrator will only measure the execution times, the throughput, and system metrics while the big data workload runs. OSv would initially be able to show vastly improved cloud bursting capabilities. However, these would require additional effort and are out of scope for this initial demonstrator.

5.2.3 Communication: vRDMA

Communication via vRDMA in our demonstrator will revolve around intra-host communication.

In MIKELANGELO vRDMA targets vastly improved I/O performance for communication between VMs based on vRDMA. The concepts around our communication prototypes include the bypassing of the networking stack inside VMs, use of shared memory for communication, and the use of RDMA-capable hardware. For intra-host setups the shared memory approach using IVSHMEM yields improved results when compared to standard KVM [4]. However, for inter-host communication our first vRDMA prototype will require RDMA-capable network interface cards. Typically Infiniband is RDMA-capable, which is not a typical interconnect for cloud computing, but rather for HPC. Thus, our initial deployment will focus on demonstrating the speedup obtained by our vRDMA prototype for intra-host communication. The scenario of running multiple VMs inside one host makes even more sense when using OSv, Especially with applications that rely on IPC for parallelism, vRDMA becomes a great way to improve the performance.

5.2.4 Monitoring: snap

The initial demonstrator will use snap to verify its functionality in a realistic, albeit low-scale, scenario.

The goal of the demonstrator is to capture the performance of HiBench using different infrastructure configurations. Using snap will first of all facilitate the capturing of system metrics. These will be used to complement the performance metrics obtained by HiBench

itself, which we will also capture. Such a configuration allows the plotting of evolving benchmark metrics and system metrics live. To achieve the integration of HiBench and snap, modifications to HiBench itself may be required. This modification complicate the implementation, however.

The planned integration with snap for the first demonstrator serves as proof of concept of snap. The same work could be carried out by using InfluxDB and Grafana, but would require significant development and configuration effort to manually expose hardware, operating system, and other system metrics from across the testbed. In fact, snap will use InfluxDB and Grafana for data storage and visualisation. The first direct results of MIKELANGELO's monitoring effort will show the use of snap plugins to capture data from various hardware and operating system sources, libvirt, sKVM, and OSv. Future iterations of this report and demonstrator will show advanced features that could not be achieved without snap. These features will include massive scalability, adaptive monitoring, and data analysis, such as anomaly detection.

5.3 Benchmarks

The initial demonstrator for the cloud integration will use benchmarks to showcase its capabilities. To provide a realistic reflection of the performance of the integration, we will use a big data benchmark. The main reason for this is a synergistic alignment with the work carried out in work package 6, where big data benchmarks will be used initially. It comes naturally to use on of the use cases as a benchmark. Of the four use cases three deal with showcasing I/O performance. The cloud bursting case is more specialised. The two HPC use cases fit better with HPC clusters, while the big data use case provides the best match for cloud computing. Furthermore, the big data use case is the only one covering benchmarks, which in turn cover a broad spectrum of workloads, spanning CPU-intensive and I/O-intensive workloads.

We present our three comparative experiments. All three experiments run the same software and the same workloads. Only their infrastructure configuration changes. All experiments will run on either a vanilla distribution of Hadoop or the Cloudera distribution of Hadoop. The latter is preferred since it offers an easier installation and advanced management capabilities. However, we need to install the whole system with identical versions on three different platforms. Thus, we expect the setup without the Cloudera additions to be simpler. As benchmark we will use the current stable version of HiBench. The reasons for choosing HiBench are laid out in D2.7 [17].

The first experiment covers a traditional cloud stack. This stack represents the current state of the art. In fact, this stack is a typical configuration as run in many data centres, such as

GWDG. The stack consists of Ubuntu 14.04 LTS as host OS, KVM, and Ubuntu 14.04 as guest OS. For monitoring we are going to use snap since it is not expected to influence to performance of our experiments. This experiment is similar to that carried out in D2.17. The results of this experiment will serve as a baseline for improvement and thus a lower boundary.

The second experiment will use a container-based configuration as cloud stack. In the first release of our demonstrator this stack will consist of Docker for virtualisation and management of the big data system. We expect the container-based configuration to provide near-native performance. To achieve this performance, the benchmark will avoid writing data to the union filesystem and use folders on the host system directly. This experiment will likely provide the best performance of all three experiments. Thus, the results will provide an upper boundary.

The third experiment will use the MIKELANGELO cloud stack with sKVM and OSv. While sKVM will be used for all VMs, OSv will be used wherever possible. A major hurdle of performing these experiments and of running completely on the MIKELANGELO stack will be the installation of Hadoop and possibly the Cloudera tools on OSv. The goal is to run as many of the relevant components on OSv, especially those that rely on fast I/O. As in the other experiments, we will use snap to monitor the performance of the experimental workloads.

6 Conclusions and Future Work

This report describes the initial integration of the MIKELANGELO components for virtualisation and cloud computing within a unified cloud computing architecture. While other parts of the project work on the underlying technologies such as the hypervisor and the guest OS, this workpackage leverages those components to improve cloud computing.

The approach in this workpackage is to integrate the results regarding virtual infrastructure technology with service management tools in a dominant cloud stack. This report is analogous to the integration of the MIKELANGELO components with HPC. Since this is the first report we first define our goals and devise a work plan. We then discuss how individual components can be used to reach those goals. Furthermore, we research options for a cloud computing framework as basis for integration. Another concern of our work is the quality assurance, which receives special attention in such a large project. Spanning a range from the host kernel to big data applications running a virtualised cluster, the cloud integration becomes a major feat in systems engineering. Thus, we dedicate a significant portion of our work and this report to setting up continuous integration.

Being the first iteration of this report, after only six month of work, this report contains a high ratio of conceptual work. However, we also describe concrete technical work being carried out for the demonstrator at the beginning of the second project year. This demonstrator will integrate key components of MIKELANGELO in a cloud computing framework for the first time. The concrete results include the choice of OpenStack as cloud stack, the integration of MPM for application management, snap for monitoring, and initial results of deploying a working testbed with continuous integration.

In addition to configuration and deployment work, the second project year will see the advent of resource optimisation algorithms. These algorithms will offer cross-layer optimisation between sKVM, OSv, and OpenStack. Workloads and signalling from the application layer will be considered additionally.

MIKELANGELO

References

[1] MIKELANGELO Report: D2.13, http://www.mikelangelo-project.eu/deliverables/deliverable-d2-13/

[2] http://unionfs.filesystems.org/

[3] MIKELANGELO Report: D5.7, https://www.mikelangelo-project.eu/deliverables/deliverable-d5-7/

[4] MIKELANGELO Report: D3.1, https://www.mikelangelo-project.eu/deliverables/deliverable-d3-1/

[5] MIKELANGELO Report: D4.4, https://www.mikelangelo-project.eu/deliverables/deliverable-d4-4/

[6] https://github.com/intel-hadoop/HiBench

[7] MIKELANGELO Report: D4.7, https://www.mikelangelo-project.eu/deliverables/deliverable-d4-7/

[8] MIKELANGELO Report: D2.19, https://www.mikelangelo-project.eu/deliverables/deliverable-d2-19/

[9] MIKELANGELO Report: D3.1, https://www.mikelangelo-project.eu/deliverables/deliverable-d3-1/

[10] https://github.com/apache/cloudstack/pulse

[11] https://github.com/eucalyptus/eucalyptus

[12] https://github.com/OpenNebula/one/pulse

[13] https://github.com/openstack/nova/pulse

[14] http://docs.openstack.org/developer/devstack/

[15] MIKELANGELO Report: D2.4, https://www.mikelangelo-project.eu/deliverables/deliverable-d2-4/

[16] MIKELANGELO Report: D2.10, https://www.mikelangelo-project.eu/deliverables/deliverable-d2-10/

[17] MIKELANGELO Report: D2.7, https://www.mikelangelo-project.eu/deliverables/deliverable-d2-7/

[18] http://superuser.openstack.org/articles/creating-openstack-debs-for-everyday-operators

[19] https://www.openstack.org/user-stories/

[20] http://opennebula.org/users/featuredusers/

[21] https://www.openstack.org/community/

[22] http://opennebula.org/community/mailinglists/

[23] https://forum.opennebula.org/

[24] https://github.com/cloudius-systems/capstan

[25] https://github.com/intelsdi-x/snap

[26] http://www.geektantra.com/2014/10/opennebula-vs-openstack-vs-cloudstack/

[27] http://archives.opennebula.org/software:rnotes:rn-rel4.4

[28] http://opennebula.org/comparing-opennebula-and-openstack-two-different-views-on-the-cloud/

[29] https://gigaom.com/2013/05/31/heres-why-cern-ditched-opennebula-for-openstack/

[30] https://jenkins-ci.org/

[31] http://docs.openstack.org/infra/zuul/

[32] https://www.rabbitmq.com/

[33] http://osv.io/

Appendix

Table 1: Feature comparison of cloud middleware

Feature	OpenNebula	OpenStack	Eucalyptus	CloudStack
General Features				
Version	4.14	Liberty (2015.2.0)	4.2.0	4.6.0
Development Language	Java, Ruby	Python	Java, C, Python, Perl	Java
Supported DBMS	mysql, sqllite	Drizzle, Firebird, Microsoft SQL Server, MYSQL, Oracle, PostgreSQL, SQLite, Sybase	PostgreSQL	MYSQL
Supported protocols and backend	LDAP, Local Users	LDAP, Kerberos, Local Users	Local users, IAM	LDAP, local users
Computation Features				
Supported Hypervisors	KVM, XEN, VMWARE, Docker	KVM, QEMU, XEN, ESXi/VC, Hyper-V, PowerVM, Docker	ESXi, KVM	VMware, KVM, Xen
Scheduling Methods	Preconfigured + Customize Policy	Filters, Weights, random, customized	EuQoS	-
Networking Features				

Feature	OpenNebula	OpenStack	Eucalyptus	CloudStack
IP v6 management	Yes	In Progress	No	No
Projects Traffic Isolation	VLAN	FLAT, FLAT DHCP, VLAN DHCP	VLAN	VLAN
Remote Desktop access protocols	VNC (noVNC), Spice	VNC (noVNC), SPICE	VNC	VNC
VM Images Features				
Supported disk formats	Raw, QCOW2, VMDK, ISO, VDI, AKI, ARI, AMI	Raw, VHD, VMDK, VDI, ISO, QCOW2, AKI, ARI, AMI	Raw, VMDK, VDI, QCOW2, ISO	QCOW2, VHD, VMDK
Supported container formats				
Supported backends	Local Storage, NFS, Ceph	Local disc, NFS, Swift, S3, Ceph	Posix	NFS, Swift, S3
Image caching	Yes	Yes	Yes	No
Create Image from VM snapshot	Yes	Yes	No	Yes
Block Storage Features				
Supported backends	NFS, LVM, Ceph, GlusterFS	NFS, LVM, Ceph, GlusterFS, Customize	NFS, DAS, EBS, iSCSI	iSCSI, NFS
Clone Volume	Yes	Yes	Yes	No
Object Storage Features				
Backend	Ceph	Swift, Ceph, S3	EBS, Walrus	Swift, S3

Feature	OpenNebula	OpenStack	Eucalyptus	CloudStack
Access method	Rest	Rest, Python-Client		Just for VM Images, Snapshots
Segmentation support (zones, groups)	Zones, Rings, Regions	Zones, Rings, Regions	No	Zones, Rings, Regions
Authentication integration support	no found	Apache 2.0	GNU	Apache 2.0
Web UI integration	no found	Yes	Yes	No
General Management Features				
Licensing	Apache 2.0	Apache 2.0	GNU	Apache 2.0
REST API	Yes	Yes	Yes	No
User Accounts				
permission granularity	Groups, Users	Tenant, Users	Groups, Users	Domain, Account, Domain Administrators, Projects
User with multiple projects or groups	Yes	Yes	No	Yes
Security				
Least privileged access design	Yes	Yes	Yes	Not Found
Fine granularity permission define method	no Found	JSON	JSON	On Web UI
centralized permission control	Yes	No	No	On Web UI
Resource Allocation				

Feature	OpenNebula	OpenStack	Eucalyptus	CloudStack
VM block provisioning	Yes	Yes	Yes	not found
Orchestration				
Complex architectures definition support	Yes	Yes	Yes	No
Autoscaling Support	Yes	Yes	No	Yes
Web UI integration	Yes	Yes	No	No
Architecture definition format	Flow	Heat	Ansible	No
Infrastructure Segregation				
Logic division	VDC, Zones	Availability Zones, Host Aggregates	Availability Zones	Zones, Pods, Clusters
Physical division	not found	Cells, Regions	Regions	Regions
Quota Definition				
Volume quantity	Yes	Yes	Yes	Not Found
# of Floating IPs	Yes	Yes	Yes	Not Found
# of security rules	Yes	Yes	Yes	Not Found
Monitoring				
Web UI integration	Yes	Yes	No	Yes
Centralized module	Yes	Yes	No	Yes
Alarms definition support	Yes	Yes	No	Not Found
Computational resource monitoring	Yes	Yes	No	Yes

Feature	OpenNebula	OpenStack	Eucalyptus	CloudStack
Networking resource monitoring	Yes	Yes	No	Yes
Images monitoring	Yes	Yes	No	No
Object Storage monitoring	No	Yes	No	No
Volumes monitoring	Yes	Yes	No	Yes
Engergy monitoring	No	Yes	No	No
Visualizations included	Yes	Yes	No	Yes
Extendible (new metrics)	not found	Yes	No	No