

Aapeli Turkki

Migrating Weather Production to External IaaS

Helsinki Metropolia University of Applied Sciences

Master's Degree

Information Technology

Master's Thesis

09 May 2017

Author(s) Title	Aapeli Turkki Migrating Weather Production to External IaaS
Number of Pages Date	49 pages 09 May 2017
Degree	Master's Degree
Degree Programme	Information Technology
Instructor(s)	Ville Jääskeläinen, Principal Lecturer Mikko Partio, Head of Group
<p>Demands as to the accuracy of weather prediction services are growing due to that the climate changing rapidly. With the help of various weather services it is possible to help for example governments and businesses to prepare for incoming storms, flood and other weather phenomena. In order to be able to provide such services, there is a growing need to store, transfer and process huge data sets.</p> <p>This thesis investigates the best practices to deploy, configure and build a scalable post-processing environment to an external Infrastructure as a Service (IaaS). The case company, Finnish Meteorological Institute, is looking at options to use external IaaS to expand the current weather prediction environment, since the current data transfer speeds are not enough to transfer needed data sets to the case company's premises on time. The case company is also looking at IaaS as a cost effective option to get on demand processing capacity when needed.</p> <p>The study goes through well-known open source technologies used to build a post-processing environment. The theoretical part also explains the IaaS architecture and how it differs from on-premise infrastructure</p> <p>The practical part explains the steps needed to build a post-processing environment to an external IaaS, as well as creating a test plan and testing phases. The study also presents what was needed to integrate a post-processing environment to part of a current weather production environment.</p> <p>The outcome of this thesis is a scalable post-processing environment on an external IaaS. The study provides necessary information for what is needed to build a post-processing environment</p>	
Keywords	KVM, Linux, SELinux, NFS, IaaS, Cloud Computing, Scalability, API, REST

Contents

Abstract

Table of Contents

Acronyms

1. Introduction	1
1.1 Current Infrastructure	2
1.2 Business Problem	3
1.3 Objective	4
1.4 Content and Scope of Study	5
1.5 Research Design	7
1.6 Structure	7
2. Used Technologies	8
2.1 IaaS	8
2.2 Ansible	10
2.3 KVM	12
2.4 NFS	14
2.5 Python	16
2.6 ClouSigma API	16
2.7 ecFlow	18
2.8 SELinux	20
3. Current Masala Production Infrastructure	23
3.1 Infrastructure	23
3.2 Production Data Flow	24
4 Requirements and Design for IaaS Environment	27
4.1 IaaS Environment Design	28
5. Deployment	31
5.1 Installing and Configuring Servers for IaaS Environment	31
5.2 Python Script for Interacting with CloudSigma API	33
5.3 Ansible for Automated Tasks	37

6. Testing	40
6.1 Creating Test Plan	40
6.2 Testing CloudSigma API Performance and Reliability	40
6.3 Testing CloudSigma Scaling Performance	41
6.4 Testing Processing with Data Sets	41
6.5 SLA Violations	42
7. Integration to Current Production Environment	43
7.1 Analysis of Post-processing Environment Test Results	43
7.2 Changing Firewall Rules for Production Phase	43
7.3 Integration to Production Environment	44
8. Discussion and Analysis	46
9. References	48

Acronyms

FMI	Finnish Meteorological Institute
ECMWF	European Center of Medium range Weather Forecasting
IaaS	Infrastructure as a Service
KVM	Kernel Virtual Machine
ASCII	American Standard Code for Information Interchange
REST	Representational State Transfer
API	Application Programming Interface
NFS	Network File System
OpenSSH	Secure Shell
IFS	Integrated Forecasting System
RMDCN	Regional Meteorological Data Communication Network

1. Introduction

The case company, Finnish Meteorological Institute (FMI), is a weather service agency and research institute. The case company acts under the Finnish Ministry of Transport and Communications. The case company has two sites in Finland, the main site is located in Kumpula, Helsinki and the second is located in Sodankylä. The Kumpula site is the main site where almost all weather prediction and meteorological services are done, the Sodankylä site is used mostly for arctic research and satellite services. The case company's main objective is to provide weather information from Finland and areas surrounding Finland. Also a big part of FMI's work is in research. The case company also provides various weather related services such as military and civil aviation weather, marine weather, ice service and road weather.

To be able to provide these services on real-time there is need for reliable infrastructure services such as network, storage, servers etc. The data used for different weather related end products can be different depending on the end product, for example data can be ASCII files or huge RAW Satellite images. Satellite images and various data sets can be quite massive, sizes of the files are related to the area which they cover. Data sets for weather products are usually raw, this means that the data sets are collected straight from different instruments such as weather stations and satellites. Raw data needs to be transferred before it can be processed to a product. Different weather services use different mathematical models to process end products. Working with big data sets and real-time services do demand a lot of capacity from the infrastructure. The case company's data sets are also available for the public as a part of INSPIRE Directive. [14]

The case company has various internal units and the present study was done for Weather Production Services (STU). STU provides infrastructure and production services also for FMI's other internal units.

1.1 Current Infrastructure

The case company provides all infrastructure services as an in-house service. The case company has to be able to provide fast network connections to save and retrieve the data, enough server capacity for processing weather products and to provide enough fast storage to store raw data and end products. The case company's current infrastructure consists of the following physical parts; Fiber Optic Network, Storage area network (SAN), Firewalls, Data storage, x86_64-Servers, VMware Virtualization, KVM Virtualization and Cray XC30 High Performance computers. Infrastructure is used both for production and research. Figure 1 illustrates an overview of the current infrastructure between ECMWF and FMI.

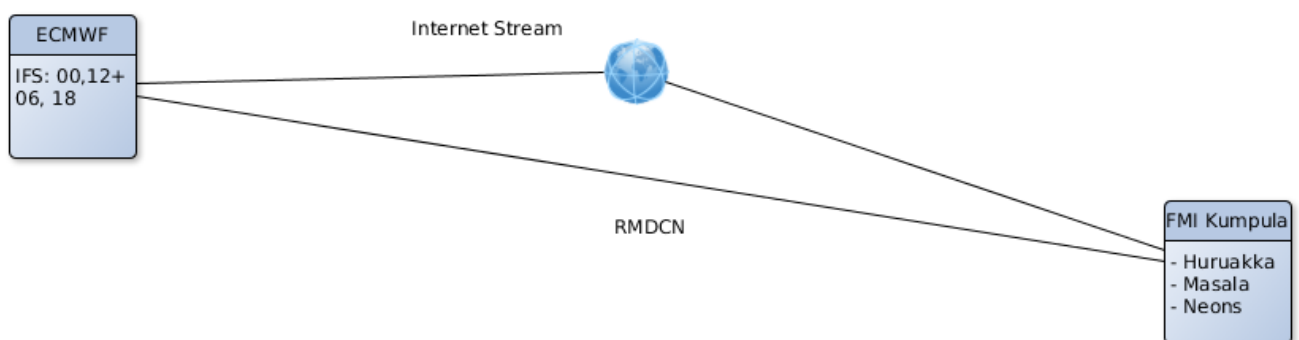


Figure 1. Overview of current infrastructure between ECMWF and FMI

Figure 1 shows an overview of the current infrastructure between European Center of Medium range Weather Forecasting (ECMWF) and FMI. ECMWF produces an Integrated Forecasting System (IFS) model four times a day; 00 UTC, 06 UTC, 12 UTC and 18 UTC. Data from these models are streamed nearly on real-time to FMI's premises. The main transfer channel goes through the Internet but, in problematic situations Regional Meteorological Data Communication Network (RMDCN) is used, RMDCN lines throughput is limited and due to this it cannot handle all the traffic. Data stream is received to the Masala-cluster where post-processing is done. When post-processing is ready pointers to the products are added in the Neons database and after

this the data is available for end customers through the Huruakka-cluster. The Masala-cluster and the Huruakka-cluster both consist of x86_64-servers and virtualized servers.

1.2 Business Problem

As the technology is making it possible to build more accurate sensors and instruments it is possible to get more detailed environmental data. With this kind of data it is possible to provide more accurate weather forecasts and other scientific data. More detailed datasets do mean bigger data size, which means that there is a growing need for capacity from infrastructure services, such as data transfer, storage capacity and computing power.

The case company's current infrastructure is able to handle current workloads efficiently, but there is a need to look for other solutions because data amounts are getting much more bigger in the future. Currently the biggest "bottle neck" is the data transfer rates between ECMWF and FMI. ECMWF provides various datasets to FMI, and from these datasets FMI produces different weather and scientific products.

Data provided by ECMWF must to be post-processed in real-time, so that the data would be available for end users on time. However, the current data transfer rates between ECMWF and FMI cannot transfer new data sets on time. ECMWF has made an agreement with Cloud Sigma that ECMWF will store all weather model data to Cloud Sigma's premises, where it is available for ECMWF's customers. The weather model data provided by ECMWF is transferred to Cloud Sigma's IaaS (Infrastructure as a service Platform) by a highly optimized Internet connection.

1.3 Objective

The goal of this thesis was to design and build a working on-demand post-processing environment which can be integrated to the case company's weather production environment. This is the first time an IaaS platform is used as a part of FMI's weather production. It is known that in the future there will be a need for this kind of environments in other FMI production chains. Figure 2 illustrates an overview of the desired infrastructure.

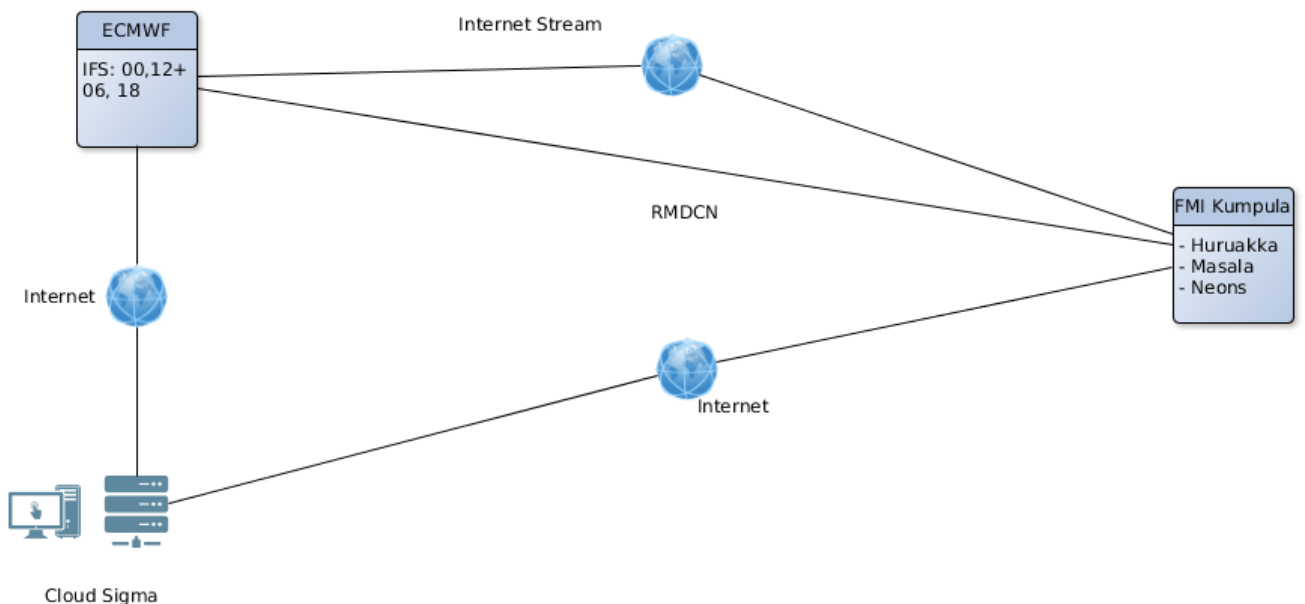


Figure 2: Overview of desired infrastructure

Figure 2 illustrates how the Cloud Sigma instance is implemented as a part of the current infrastructure. The existing connections and chains will stay intact and the Cloud Sigma part will be added as a new element. Global data sets will be post-processed at Cloud Sigma IaaS and transferred to FMI Neons database.

The case company is also looking for options to use different cloud providers in that case that the Cloud Sigma cannot provide needed capacity or Service Level Agreement (SLA). To enable easy migration between different cloud providers it is important to use well

known open source technologies to and make sure to minimize the use of cloud provider specific technologies and to make portable configurations and setups.

To be able to design and build this kind of an environment to an external IaaS, the key technologies used in IaaS platforms have to be understood. It is also important to understand that what kind of requirements the post-processing environment has, how the data is transferred to the servers which are doing post-processing and how processed data is transferred to the end users.

The key technologies which were needed to build a post-processing environment are listed below:

- Infrastructure as a Service (IaaS)
- Ansible
- Kernel Virtual Machine (KVM)
- Network File System (NFS)
- Python
- CloudSigma API
- ecFlow
- SELinux

These technologies are explained in detail in Chapter 2.

1.4 Content and Scope of Study

This thesis explains what is needed to be able to design and build an on-demand post-processing environment to an external IaaS system as a part of an existing weather production chain.

The design of the post-processing environment needed to use the best practices of cloud computing and well known open source technologies. Environment had to be easily exportable to other cloud providers' environments, with minimal need of extra work and to avoid vendor lock-in. Documentation had to be up-to-date and exact so any System

Administrator with Linux knowledge could easily setup the same kind of environment to other cloud infrastructure.

This study had four main tasks; designing post-processing environment, building the designed environment, testing the built environment and integrating the new post-processing environment as part of current weather production environment.

The design was presented to the projects steering group, which approved the used technologies and the designed post-processing environment. After the design was approved the new environment was built based on the design.

The testing phase started after the environment was ready and the testing was made with two experts from STU's production services. After testing a phase meeting was held to see if there was a need of any improvements. After required improvements were done, the post-processing environment was integrated as a part of the current weather production environment.

Features out of the scope of this study were monitoring the external IaaS environment and deploying the built environment to another cloud providers IaaS. Monitoring external IaaS is an important part of a 24/7 production environment, any problem should be noticed immediately when it arises. Defining specifications and configuration for monitoring solutions for external IaaS was left for another project in the case company. Deploying the post-processing environment to another cloud providers IaaS will be a new project in the case company when there is more experience from the Cloud Sigma project.

The outcome of this study should be easily reusable in different IaaS solutions. The theory part explains how to use well known open source technologies in Cloud Computing. The ecFlow is the only solution which is not reusable since it is not publicly available, but there are other job control applications publicly available which have the same functions.

1.5 Research Design

A process flowchart was needed before it was possible to start designing the post-processing environment. For this several meetings were held with the experts from the STU's production unit. Two different process flowcharts were produced, one with a brief overview and one more detailed process flowchart which can be easily updated when the processing environment changes. Analyzing of the technologies used in the current weather production environment were needed. With the information from the analysis of current weather production environment it was possible to research certain open source technologies and to define which protocols and features are required.

The design was presented and approved in the case company. The post-processing environment was built based on the design. A testing phase started immediately after the post-processing environment was ready. The testing phase included various parts which all needed to function before the integration could be done to the current weather production environment.

1.6 Structure

This study has eight chapters. Chapter 1 gives an overview of the study. Chapter 2 goes through well-known open source technologies used in the study. It also describes protocols and their configurations. Python code for API attributes is also described in Chapter 2. Chapter 3 describes the current infrastructure and its data flow. Chapter 4 presents the post-processing environment design. Chapter 5 explains the steps to deploy and configure the post-processing environment on Cloud Sigma IaaS. Chapter 6 covers the testing phase. Chapter 7 covers the integration process to the current weather production environment. Chapter 8 presents the discussion and analysis.

2. Used Technologies

This chapter provides a technical understanding of the key technologies needed to build a scalable post-processing environment to an external service providers IaaS.

2.1 IaaS

IaaS is one of the three service models used in cloud computing, other two are PaaS (Platform as a Service) and SaaS (Software as a Service). PaaS and SaaS are not covered in this study. IaaS service model is presented in Figure 3.

IaaS is hosted and maintained by the cloud provider including all hardware components such as servers, data storage and networks. Usually IaaS is distributed into multiple data centers and different geographic locations. Running IaaS on multiple data centers and different geographic locations removes single point of failure and makes it possible to provide good SLA (Service Level Agreement). Cloud provider offers virtualized hardware components such as; servers, storage and networks to the end customer. Virtualized hardware is usually running on multiple hardware so if some hardware components fail it does not affect the provided services. IaaS is often accessed through the public Internet. Some cloud providers do offer VPN-services (Virtual Private Network) or MPLS (Multiprotocol Label Switching), these services are used when companies need to scale-out their services to external IaaS.

IaaS gives users an “infinite” amount of resources with zero investment to real hardware. New resources are available immediately when needed and provisioned resources can be removed automatically when none of the resources are used. This makes resource usage optimal, no resource are used if it is not necessary. Resources are used on demand and the customer only pays for the resources which are used.

Cloud providers customers are able to self-provision their own infrastructure by using provided tools which usually are web-browser based GUI (Graphical user interface) or API (Application programming interface). API is used when automation is required. API is an effective way to self-provision resources if necessary. API usually has all the same

features which are available in the web-browser based GUI. The basic principles of API are covered later in this chapter.

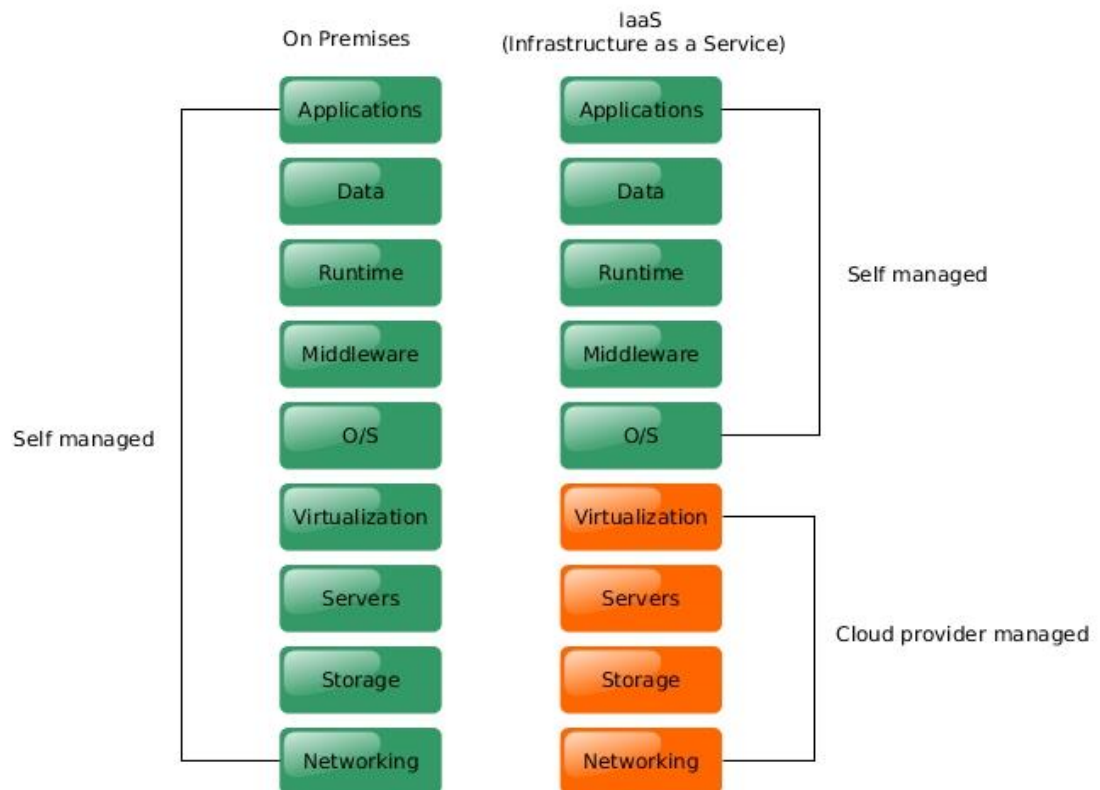


Figure 3: Difference between on premises hosting and IaaS

Figure 3 illustrates differences between on premises hosting and IaaS. Cloud provider is responsible of the hardware and virtualization on IaaS service model. This gives customer ability so scale out their applications and services, without need to worry about the hardware resources.

2.2 Ansible

Ansible is an agentless IT automation tool, which can be used to manage a large number of servers, applications etc. Ansible does not need any extra applications to work, such as agent, security certificates etc.

Ansible uses SSH to connect to servers and run given tasks. Since SSH is used it is good practice to use SSH-keys so there is no need to enter password every time and SSH-keys also give better security. Ansible transfers content over SSH before running task, when task is ready content will be deleted.

Ansible inventory is by default presented in simple text files. The managed machines are divided into groups, there is no restrictions for group names and a managed machine can also belong to multiple groups. An example of an Ansible hosts file is presented in Listing 1.

```
[himan]
himan

[himan-compute]
himan-compute
```

Listing 1: an example of Ansible hosts groups file

Listing 1 shows an example how groups are listed in the Ansible hosts file. [himan] is the group name and after that comes the host name of managed machines.

Ansible has its own language for describing orchestration and configuration files. Playbooks are written in YAML language. Playbook can have multiple tasks “plays”. An example of Ansible playbook is presented in Listing 2.

```

---
- hosts: himan
  sudo: false

  tasks:
    - name: Generate fmi.repo metadata.
      shell: createrepo --update -v /var/www/html/RPMS/
      ignore_errors: true
      register: metadata

```

Listing 2: an example of Ansible playbook to generate RPM (RPM Package Manager) repository metadata

Listing 2 shows how to generate RPM repository metadata. “hosts” determine to which group task will be done. “sudo: false” means that no sudo command is used to issue the commands. “tasks:” have multiple parameters:

- name: Describes what the playbook does
- shell: shell command is issued on the target host
- ignore_errors: continue if error occurs, by default no more steps are performed if failure occurs.
- register: register the value of the previous step, registered value can be used as a variable. Registered value is valid until playbook run is finished.

YAML language used in Ansible playbook is easily understandable, so there is no need previous coding background to be able to write Ansible playbooks. Playbooks are also portable, any Linux machine which has Ansible can run playbooks.

2.3 KVM

KVM (Kernel-based Virtual Machine) is a Linux based hypervisor. KVM is built in Linux kernel module since Linux 2.6.20. KVM is a full virtualization solution for X86 hardware. KVM can be run only by x86 hardware which include x86 CPU (central processing unit) extensions Intel VT and AMD-V. Kernel module (kvm.ko) provides core components for the virtualization and CPU specific module depending of CPU architecture. KVM makes it possible to run multiple virtual machines on a single x86 server. All virtual machines have private virtualized hardware such as: storage, network interface etc. KVM supports all notable operating systems such as Linux, BSD and Windows. Figure 4 illustrates the KVM architecture.

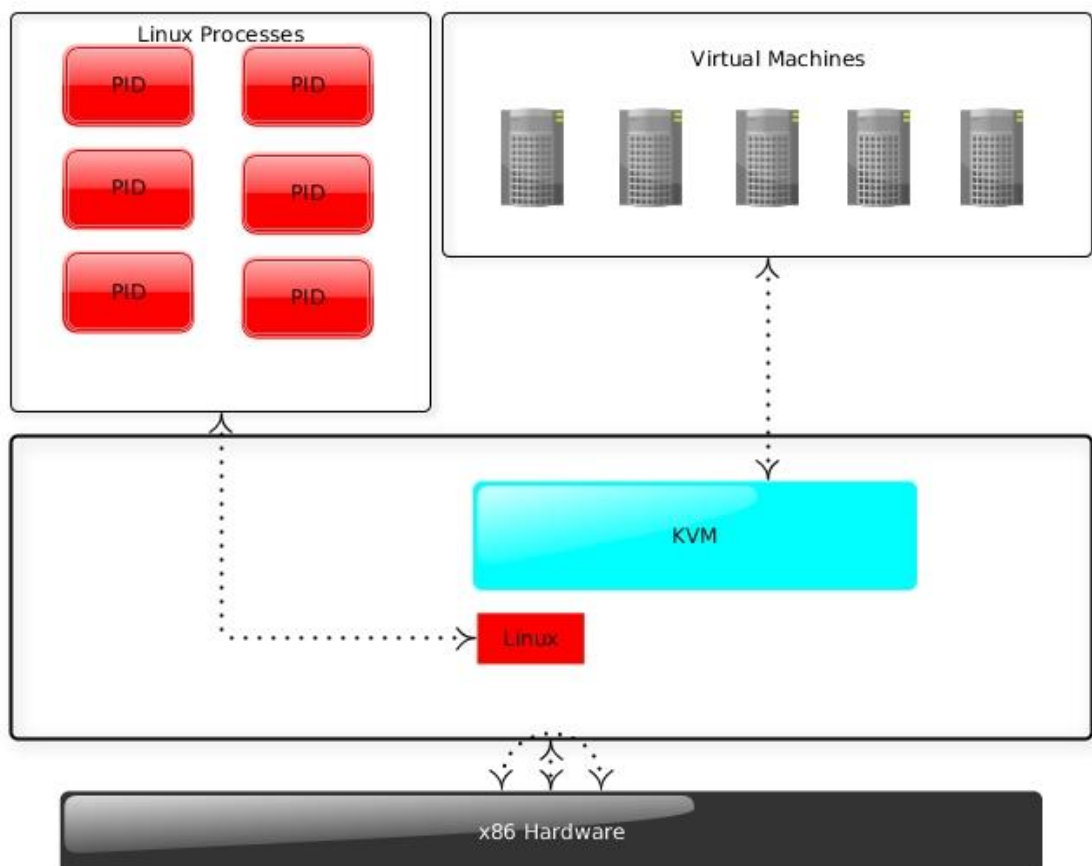


Figure 4: KVM architecture

Figure 4 shows an overview of KVM architecture. Linux runs on top of x86 hardware, and KVM runs on top of Linux. Each KVM hypervisor hosted virtual machine has its own PID (Process ID).

KVM supports cloning, snapshots and live-migration from one x86 host to another x86 host, without an effect on the virtual machine. These are the key benefits of virtualization. Cloning makes it possible to start multiple customized virtual machines on-demand and this is an important feature when building scalable environments. Snapshot makes it possible to rollback to an earlier state of virtual machine if something goes wrong or there is a risk of that something goes wrong, example when making system upgrade. Live-migration makes it possible to have “always on servers”. In case the host needs to be removed because of a faulty hardware or the maintenance, all virtual machines on that host can be migrated to another host in the same cluster without affecting the state of the virtual machines.

Managing several clusters and hosts hosting hundreds of virtual machines through CLI (command line interface) is not effective. Libvirt has been developed to make management easier. Libvirt is a software collection which makes it possible to manage KVM and other virtualization platforms such as Xen and VMware ESX. Libvirt includes an API which provides tools for managing different resources. Some examples are presented below:

- Virtual machines: Basic operations such as create, delete, start, stop, pause, restore, snapshots, migrate. Add and remove components such as CPUs, network interfaces, memory, disks etc.
- Storage management: Add raw LUNs (Logical unit number), create data storages, create file images, mount NFS (Network File System) shares etc.
- Network management: Create and configure interfaces, vlans, bridges etc.

Managing KVMs through web-browser based GUI is possible. A virtual data center manager such as oVirt will provide this feature. oVirt can manage multiple data centers

running multiple hosts with KVM and libvirt. Though oVirt it is possible to do the same operations which libvirt provides.

2.4 NFS

NFS (Network file system) is a distributed file system. NFS uses a traditional server-client model. NFS server exports a shared file system which clients mount into their local file space. Figure 5 illustrates the NFSv3 architecture.

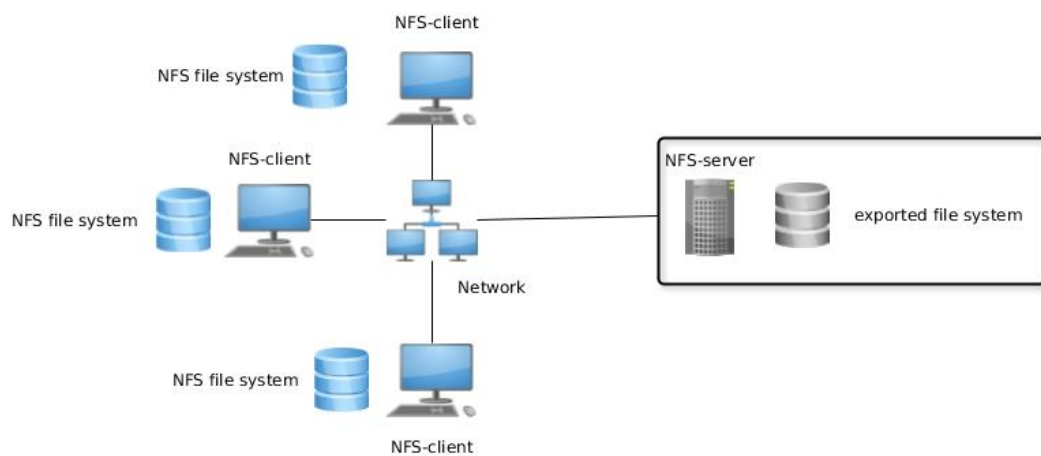


Figure 5: NFSv3 architecture

Figure 5 illustrates basic NFS architecture between NFS server and NFS clients. NFS server exports shared file system and NFS clients mount it over network.

Linux has VFS (virtual file system switch) which provides support for multiple file systems such as NFS, XFS, ext4fs and CD-ROM on a single host. When a file access request is received, VFS determines which file system is used to satisfy the request. The difference between NFS and local file systems is that I/O (input/output) requests will pass through network if I/O requests cannot be completed locally.

VFS passes a request destined for NFS inside Linux kernel, NFS translates received I/O request to correct NFS operation. NFS's various operations (READ, WRITE, RENAME, GETATTR, etc.) are documented in the NFSv3 RFC 1813. [6] Once NFS operation is

chosen from the I/O requests it is performed in RPC (remote procedure call) layer. RPC offers ways to perform procedure calls between different systems.

RPC has XDR (external data representation) layer. XDR makes sure that the NFS clients and servers do speak the same language regardless of the underlying architecture. The data presentation type can be different between the requesting host and the host answering the request. XDR converts the data type into a common representation. [5]

After the data type has been converted into a common representation, the request is transferred over the Ethernet network.

Figure 6 illustrates the NFS client and NFS server stack. [5]

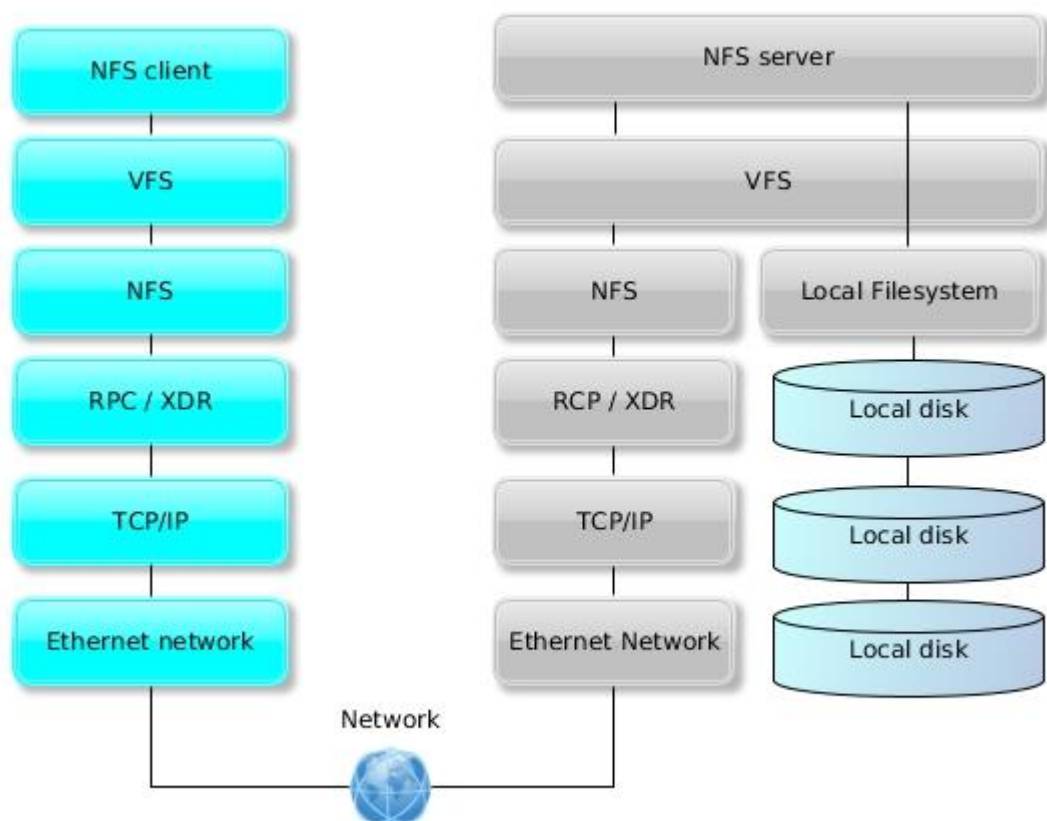


Figure 6: NFS client and NFS server stack

Figure 6 shows the different layers used between NFS client and NFS server.

NFS server receives requests through Ethernet network. The request goes through the same stack as on the client side until it reaches NFS server. NFS server has NFS daemon running which is responsible for identifying requests target file system tree. NFS is used again to access the requested file in the local file system. [5]

2.5 Python

Python was designed by Guido Van Rossum. Today Python is developed by Python Software Foundation, which has the copyrights for Python versions 2.1 and newer. [7] Python is a dynamic programming language. Python is used from small scripts to large server side applications. Python has two main branches Python 2.x and Python 3.x. Latest Python branches are: Python 2.7.11 and Python 3.5.1. [8] [9]

2.6 ClouSigma API

CloudSigma API (Application Programming Interface) uses the REST architecture. REST (Representational state transfer) is used also by web browsers. Web browsers use a set of common methods for HTTP defined in RFC 7231. [10] Figure 7 lists the RFC 7231 common methods and descriptions. [10]

Method	Description
GET	Transfer a current representation of the target resource.
HEAD	Same as GET, but only transfer the status line and header section.
POST	Perform resource-specific processing on the request payload.
PUT	Replace all current representations of the target resource with the request payload.
DELETE	Remove all current representations of the target resource.
CONNECT	Establish a tunnel to the server identified by the target resource.
OPTIONS	Describe the communication options for the target resource.
TRACE	Perform a message loop-back test along the path to the target resource.

Figure 7: RFC 7231 common methods. [10]

CloudSigma API is an important part when building an automated IaaS environment. CloudSigma API enables the same functions which the web browser based GUI has. API is used to do various actions on CloudSigma IaaS such as; stop, start and clone virtual machines. An example of simple Python API script is presented in Listing 3.

```
#!/usr/bin/python
import requests
import sys
headers = {'Authorization': 'Basic YWFwZWxpLnR1cmt1vMGwzcHBhcyExISlylg=='}
url = 'http://zrh.cloudsigma.com/api/2.0/servers/?limit=0'
r = requests.get(url, headers=headers)
print(r.status_code, r.reason)
print(r.text[:300] + '...')
```

Listing 3: An example of Python API script to list all servers by an authenticated user.

Listing 3 shows how to list all the servers on the account. The sample script uses Python module requests to perform an API call. Authentication is done with the authorization

header. The authorization header has Base64 encoded user name and password.[12] A GET request is issued with URL and header. The answer is received with status code and payload. An example answer to an API call is presented in Listing 4.

```
(200, 'OK')
{"meta": {"limit": 0, "offset": 0, "total_count": 1}, "objects": [{"name": "example", "owner":
{"resource_uri": "/api/2.0/user/0a822ff1-687e4-47f2-bead-08e2cfb4a2a0/", "uuid":
"0a822ff1-687e4-47f2-bead-08e2cfb4a2a0"}, "resource_uri":
"/api/2.0/servers/73dfe938-6ed6-4b65-9ad3-6e4e776bdb6e/"
```

Listing 4: Example answer to an API call.

Listing 4 presents an answer to an API call. The answer has the status code and requested information in JSON format.

2.7 ecFlow

ecFlow has been developed by ECMWF. ecFlow is a package that allows the running of large amount of jobs with dependencies in a controlled environment.[13] ecFlow has CLI (command line interface), Python interface and ecflowview GUI.

An introduction of ecFlow describes ecFlow as follows: *“ECFLOW submits tasks (jobs) and receives acknowledgments from the tasks when they change status and when they send events, using child commands embedded in your scripts. ECFLOW stores the relationships between tasks, and is able to submit tasks dependant on triggers, such as when a given task changes its status, for example when it finishes.”*[13] Figure 8 shows an ecflowview GUI.

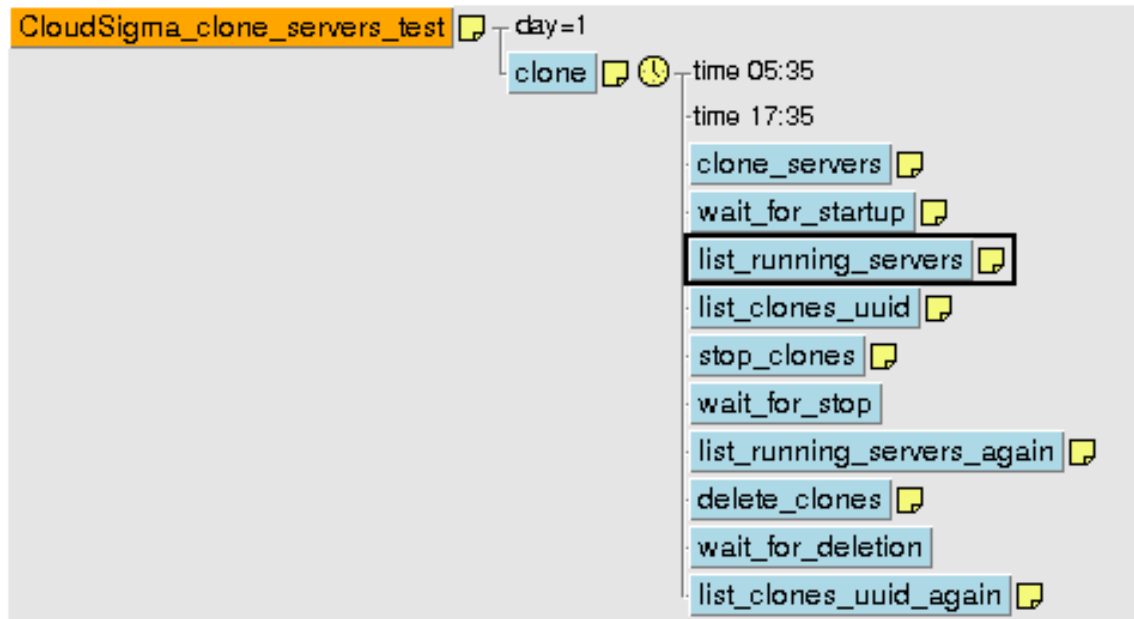
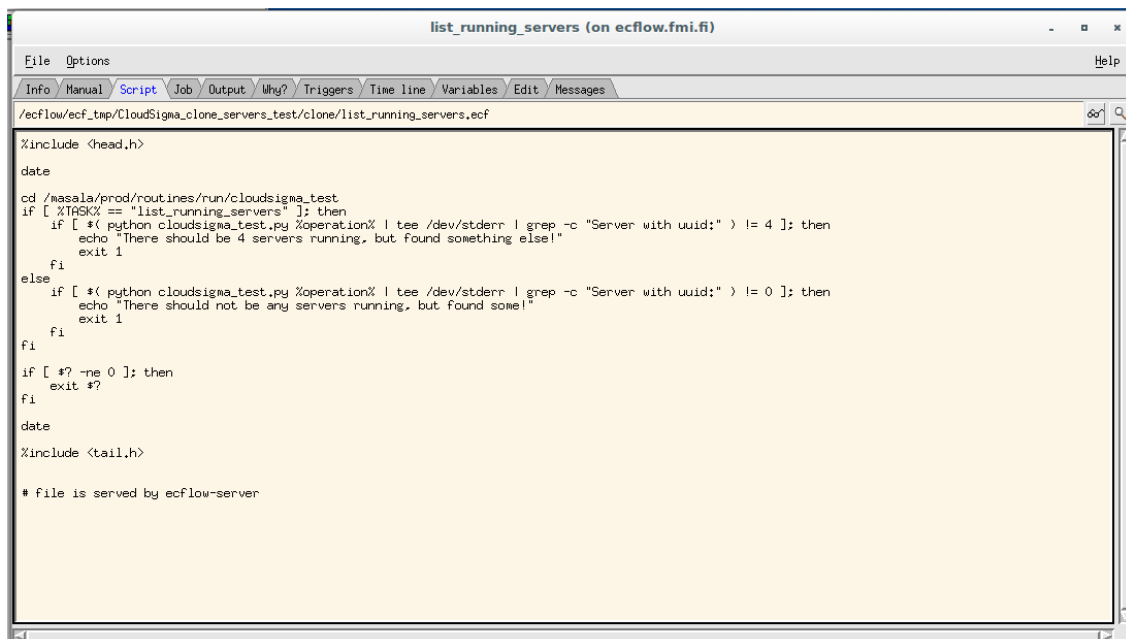


Figure 8 ecflowview GUI.[13]

In Figure 8 the ecFlow tasks are inside a “suite”, “CloudSigma_clone_servers_test” represents a suite. The clone determines what time the suite will be executed, after the execution tasks in the suite will be executed from top to bottom, if a task fails error code will be 1 and running the suite stops in to the failed task. STDOUT (Standard output) is available on the failed tasks for debugging reason. When the error is corrected, the suite can be started from the point where it failed. Figure 9 shows the ecflow script syntax.



```

list_running_servers (on ecf.flow.fmi.fi)
File Options Help
Info Manual Script Job Output Why? Triggers Time line Variables Edit Messages
/ecflow/ecf_tmp/CloudSigma_clone_servers_test/clone/list_running_servers.ecf

%include <head.h>
date

cd /masala/prod/routines/run/cloudsigma_test
if [ "${TASK}" == "list_running_servers" ]; then
  if [ $(python cloudsigma_test.py %operation% | tee /dev/stderr | grep -c "Server with uuid:" ) != 4 ]; then
    echo "There should be 4 servers running, but found something else!"
    exit 1
  fi
else
  if [ $(python cloudsigma_test.py %operation% | tee /dev/stderr | grep -c "Server with uuid:" ) != 0 ]; then
    echo "There should not be any servers running, but found some!"
    exit 1
  fi
fi

if [ $? -ne 0 ]; then
  exit $?
fi

date
%include <tail.h>

# file is served by ecf-flow-server

```

Figure 9 script syntax from “CloudSigma_clone_servers_test” suites task “list_running_servers”.

Figure 9 shows the ecf flow script syntax. The example script runs Python script “list_running_servers”, depending on the phase of the task. The script looks for 4 running servers, if more or less is found, the script will exit with code 1 and stop and if 4 servers are found the script will continue.

2.8 SELinux

The development of SELinux was started by NSA (National Security Agency). Linux kernel 2.4 family presented SELinux as a loadable kernel module. Today SELinux can be found for example in Android OS, since it provides the necessary tools to have more fine grained security policies and settings.

SELinux is an implementation of an MAC (mandatory access control). It means, for example that a process or an application has to have a policy which allows it to access a file or service. A good example of it is that there is a webserver running Apache. The website has a function for sending email which needs to use sendmail for sending email.

If there is no policy in the place that Apache is allowed to use sendmail SELinux will deny this. An example of a denied request is presented in Listing 5.

```
sendmail: fatal: chdir /var/spool/postfix: Permission denied
```

Listing 5: Example of denied request to use sendmail.

SELinux has three different operational modes; permissive, enforcing and disabled. The permissive mode means that violations will be logged but operations are allowed. The permissive mode is a good starting point when taking new services into use. One can build correct policies by auditing SELinux log-files. Enforcing mode means that violations will be logged and operations will be denied. Disabled mode means that SELinux module is not active.

SELinux has a built in “sestatus” command which can be used to show current SELinux policies. By using “sestatus” command and using grep tool one can check for example that is Apache allowed to use send mail, this is presented in Listing 6.

```
~]# sestatus -b |grep httpd_can_sendmail
httpd_can_sendmail          off
```

Listing 6: Example of sestatus command output.

Listing 6 presents an output of sestatus command with option “-b” (Display current state of booleans.) and grep httpd_can_sendmail. The output shows that Apache is not allowed to use sendmail. To change this Apache boolean needs to be changed, this can be done with “setsebool” command. Listing 7 presents usage of “setsebool” command.

```
~]# setsebool -P httpd_can_sendmail 1
~]# sestatus -b |grep httpd_can_sendmail
httpd_can_sendmail          on
```

Listing 7: Example how to allow httpd to use sendmail.

Listing 7 presents how to change Apache boolean so that Apache can use the sendmail. Option -P (all pending values are written to the policy file on disk. So they will be persistent across reboots). The output of sestatus shows now that Apache is allowed to use sendmail.

3. Current Masala Production Infrastructure

This chapter describes the current Masala production infrastructure. It also goes through the production data flow from raw data to an end product.

3.1 Infrastructure

The case company's current Masala production infrastructure consists of six bare-metal GPU (Graphics processing unit) servers and one virtual machine. This server architecture differs from the architecture used in CloudSigma. In Masala cluster processing is done with GPU and in CloudSigma processing it is done with CPU. The workload is divided between the six servers. ecFlow takes care of submitting jobs and it determines which server will start the next job. A new job will be started on the server which has the least load at that time. Figure 10 shows the current masala production infrastructure.

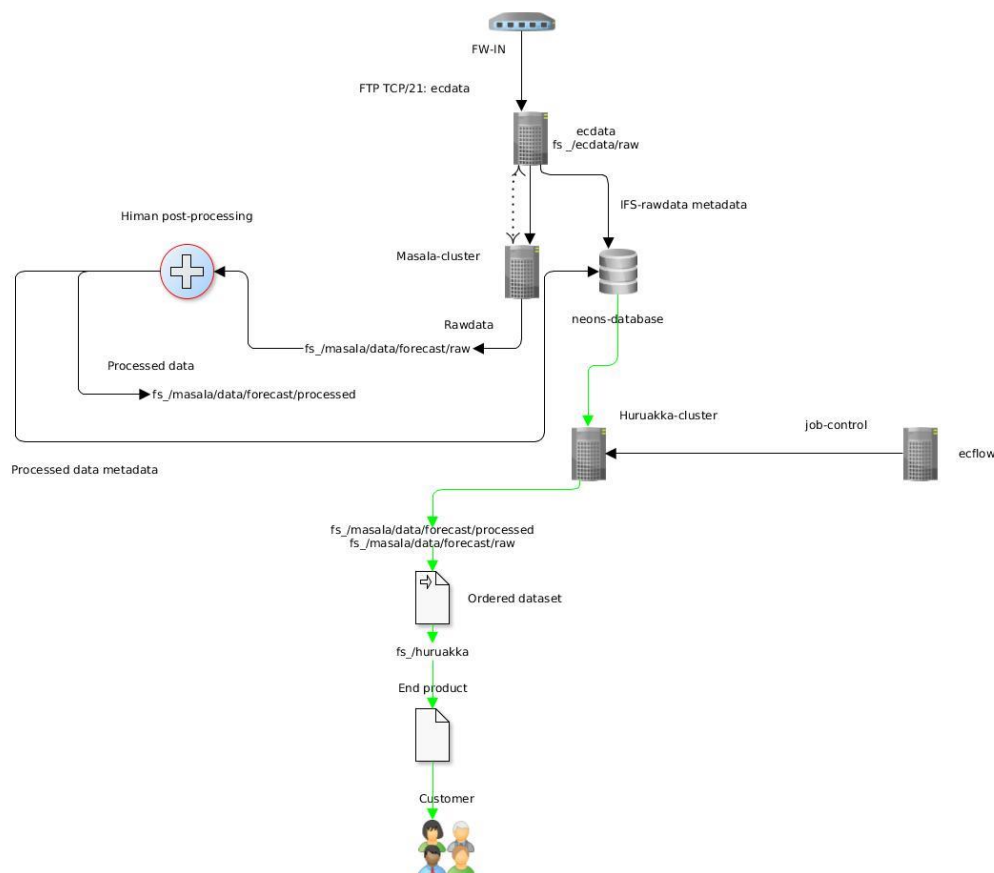


Figure 10. Masala production infrastructure.

Figure 10 ECMWF data stream is received by ecddata trough FTP-protocol. All six masala cluster members access raw data from NFS disks, output data is also written into NFS disks.

3.2 Production Data Flow

The current production chain has various steps until the product is ready to be made available for the end customers. Figure 11 shows the production data flow from raw-data to a final end product.

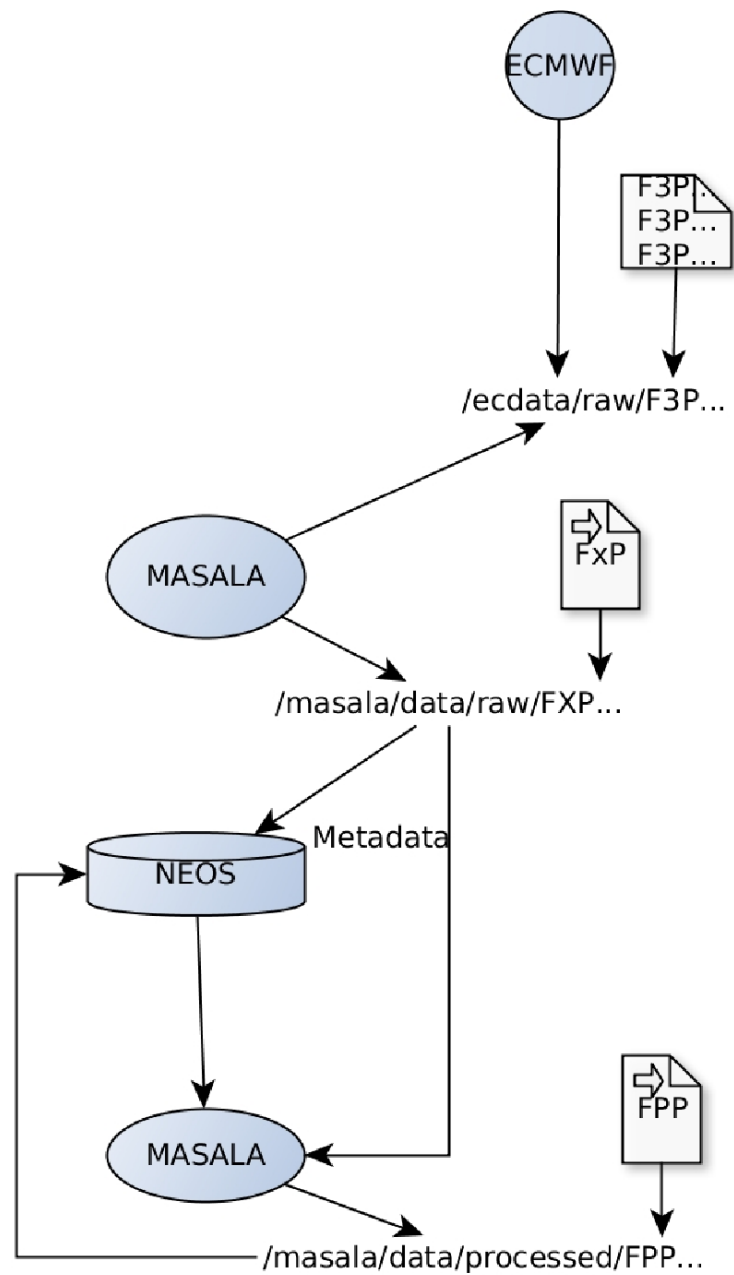


Figure 11: Masala cluster production data flow.

ECMWF streams data into */ecdata/raw* disk. The data stream starts when the first member is ready. The data stream continues until ECMWF's HPC cluster has computed the last member and that the file has been transferred to FMI.

Members of Masala cluster start to read */ecdata/raw/F3P* files and each F3P includes over 1000 F3P-lines. F3P-lines from all F3P files are then extracted to new FxP files. FxP file has only 1 FxP line, so one F3P will produce over 1000 FxP files. FxP files are then written into */masala/data/raw/FxP* disk, at the same time metadata of these FxP files is written into NEONS-database.

Processing will start immediately when the first FxP file and its metadata is available for Masala cluster members. ecFlow decides which member of the cluster starts the next job. The processed data is written into */masala/data/processed/* disk. The processed data is presented in FPP files. FPP files metadata is written into the NEONS database. After these steps customers can access data via NEONS database.

4 Requirements and Design for IaaS Environment

This chapter describes the requirements for the IaaS environment. It also presents the desired design.

The case company defined five different main requirements for the CloudSigma project; cost efficiency, portability, automatic scalability, security and that the design is not IaaS provider dependent.

To keep the cost as low as possible servers were up only when needed. Cloudsigmadb (main server) will be started 5 minutes before ECMWF starts streaming data to CloudSigma. Compute nodes are provisioned 2-3 minutes before RAW data is available for processing. After processing is done, all compute nodes are destroyed automatically and main server will be powered off when all files have been transferred to FMI.

Portability is possible by downloading cloudsigmadb and compute node KVM images via CloudSigma API and converting the KVM images into RAW format. It is possible to upload these RAW KVM images to some other Cloud Providers IaaS. Python API script is using widely used Python libraries, so it is relatively fast job to make it work with other API.

Automatic scalability was done by using API and ecFlow. ecFlow commands cloudsigma_clone.py script with the parameter of desired number of compute nodes, compute nodes are then automatically provisioned and are available to run jobs.

Security requirements were achieved by using content only from trusted sources. No passwords were used, authentication was done by using SSH-keys. Firewalls were configured on the cloudsigmadb and to the compute nodes by using iptables. Private VLAN was used for NFS traffic. SELinux was implemented on all the servers to provide extra layer of security.

4.1 IaaS Environment Design

Two CentOS 7 Linux servers, cloudsigmadb and cloudsigma-compute were created. These servers have different roles.

Cloudsigmadb is the main server which handles receiving and transferring data. It also offers other necessary services such as database, NFS, DHCP and RPM-repository. Cloudsigma-compute is compute image. Compute node clones are provisioned from this image.

Chapter 5 covers more detailed information about the deployment and final configuration. Figure 12 presents the IaaS environment design which was approved by the steering group.

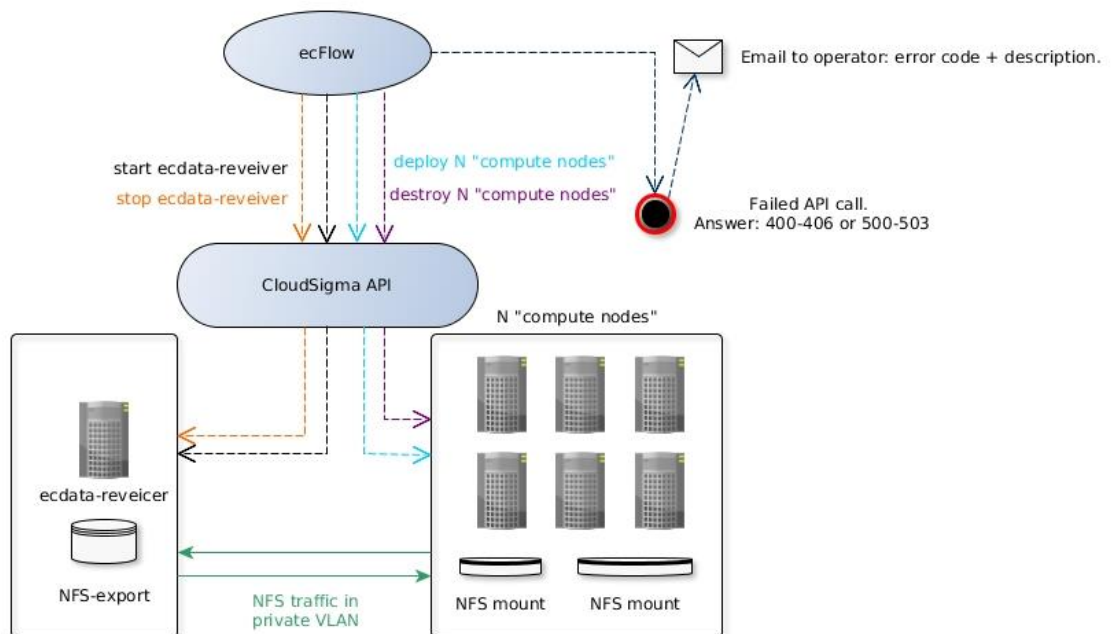


Figure 12: IaaS environment design.

EcFlow is responsible for the following tasks:

- starting and stopping servers (cloudsigmadb)

- cloning and destroying compute nodes
- distributing jobs to compute nodes

Cloudsigmadb is started via CloudSigma API by ecFlow 5 minutes before ECMWF starts the data stream. Cloudsigma.db has two NFS exports for the compute nodes which are presented to compute nodes through private VLAN, it also acts as a DHCP-server for the compute nodes.

Provisioning of compute nodes will start immediately after cloudsigmadb has been started. Each compute node mounts NFS shares on boot. After processing is done all provisioned compute nodes are destroyed, when cloudsigmadb finishes transferring processed data to FMI it will be stopped.

If an API call fails, the answer will include the error code which is then emailed to the operator with an explanation and link to Wiki. Wiki includes detailed instructions on what should be done next. The error response codes are:

- 400: Bad Request. This status means that there is an error in the request. The request error may be a data format error (non-valid JSON or XML) or an invalid value.
- 401: Unauthorized. The provided credentials are incorrect or missing. This response status is a normal part of digest authentication in which case, the response will contain WWW-Authenticate header with an authentication challenge.
- 402: Payment Required. This error means that there are not enough funds in the account to complete action. It occurs when trying to buy subscription without having enough funds in the account, or when trying to start a server without having enough funds for burst usage of 5 days.
- 403: Forbidden. The provided credentials are correct but the user is not permitted to complete the action. This status is used for either “permission” or “operation not allowed” error.

- 404: Not Found. The requested object does not exist. This error occurs when requesting a non-existing resource. The resource may have never been created, or it might be deleted.
- 405: Method Not Allowed. This error occurs, when using incorrect HTTP method on an URL. For example DELETE requests are not allowed on /profile/ URL, and will return a 405 status.
- 406: Not Acceptable. This error occurs when the content type requested through the Accept header is not supported by the API. The content types supported by the API are application/json, application/xml, and */*, which defaults to application/json. If the Accept header of the request does not contain any of this content types, a 406 status will be returned.
- 500: Internal Server Error. This status means a system error has occurred. Please contact support if you encounter such an error.
- 503: Service Unavailable. This status means that the system temporarily cannot fulfill the request. This status is returned for concurrent updates, when the client makes multiple concurrent requests which try to update the same values, or when the system is out of capacity.[17]

Error codes and descriptions listed above are defined by RFC 2616. [18]

5. Deployment

This chapter goes through the deployment process. It also presents important configurations.

Deployment was divided into three parts:

1. Installing and configuring needed servers for IaaS environment
2. Writing needed Python scripts for interacting with CloudSigma API
3. Writing Ansible-playbooks.

The three stages are introduced below.

5.1 Installing and Configuring Servers for IaaS Environment

To achieve the cost efficiency request cloudsigmadb and cloudsigma compute servers were installed on a private workstation by using the Linux virt-install tool. Listing 8 shows an example syntax to create a CentOS virtual-machine.

```
~]# virt-install \  
> -n cloudsigmadb \  
> --os-type Linux \  
> --os-variant rhel7 \  
> --memory 8192 \  
> --vcpus 8 \  
> --disk path=$HOME/KVM_DISK/cloudsigmadb.qcow2,bus=virtio,size=10 \  
> --graphics none \  
> --cdrom $HOME/Downloads/CentOS-7-x86_64-Minimal-1611.iso \  
> --network bridge:br0
```

Listing 8: syntax to create CentOS virtual-machine with virt-install.

The next step was to install and configure following services to cloudfsigmadb:

1. Apache: for serving FMI RPM-packages to compute nodes.
2. DHCP-server: offer ip-addresses for provisioned compute nodes.
3. Postgresql: for RAW and processed metadata.
4. vsftpd: to receive ECMWF data stream.
5. NFS client: configure correct exports for compute nodes.
6. SSH: add ssh-keys for root-user, compute user and disable password logins.
7. Iptables:
 1. allow SSH connections from FMI's NAT address.
 2. allow SSH connections from compute nodes.
 3. allow HTTP connections from compute nodes.
 4. allow FTP connections from ECMWF NAT address.
 5. allow Postgresql connections from compute nodes.
 6. allow NFS connections from compute nodes.

For cloudfsigma compute the following services were installed and configured:

1. NFS client: add correct mount options
2. Postgresql-client: for connecting to cloudfsigmadb
3. SSH: add ssh-keys for root-user, compute user and disable password logins.
4. Iptables:
 1. allow SSH connections from FMI's NAT address.
 2. allow SSH connections from compute nodes

After all the services were installed, configured and tested both virtual machine images were converted to RAW format so those could be uploaded to the CloudSigma environment. Listing 9 shows an example how to convert .qcow2 KVM-image to RAW format.

```
~]# qemu-img convert cloudfsigmadb.qcow2 cloudfsigmadb.raw
```

Listing 9: an example how to convert .qcow2 virtual-machine image to RAW format.

After both KVM-images were uploaded to the CloudSigma IaaS, the following extra service were bought from CloudSigma:

1. A static ip-address for both servers. Static ip-address were needed so ecFlow can connect automatically to the servers.
2. Private VLAN. As shown in the design, all the NFS traffic use private VLAN, and cloudsigmadb also offers DHCP-service on private VLAN.

Static ip-address and private VLAN were added to the configuration. Changed were done through CloudSigma GUI.

5.2 Python Script for Interacting with CloudSigma API

Two different Python scripts were needed. One for starting and stopping cloudsigmadb and cloudsigma compute servers. This script had been implemented before this project so it is not listed here. The second was for provisioning cloudsigma compute clones. This script had to provide the following functions:

1. Provision and autostart N clones from cloudsigma-compute image.
2. list provisioned clones.
3. stop provisioned clones.
4. destroy provisioned clones.

Listing 10 shows a sample of the function used to clone cloudsigma-compute.

```
def clone_servers():
    uuid = '73dfe938-6ed6-4b65-9bbb-6e4e776bdb6e'
    tags = open("cloned_servers_tags", "wb+")
    data = {'count': 10, 'auto_start': 'true'}
    data_json = json.dumps(data)
    r = api_post('/servers/' + uuid + '/action/?do=bulk_clone_start', data=data_json)
    if r:
```

```

tag = json.loads(r)
tags.write(tag["tags"][0])
else:
    return None

```

Listing 10: A sample of a function used to clone cloudsigma-compute.

Listing 10 shows how clone_servers function works:

- uuid points to cloudsigma-computeimage from which clones are provisioned. All items in IaaS environment such as servers do have their unique id in a database. All actions through API need to be done by using these uuids.
- tags opens a file called “cloned_servers_tags” where compute instance uuid is written, all cloned servers can be listed by using this uuid.
- data has two values:
 - how many servers will be provisioned.
 - auto-start cloned servers.
- data_json gets data values in json formatted string.
- r uses api_post function to send API call to provision clones.
- if r gets accepted answer json payload (uuid) is written into “cloned_servers_tags” file.
- if r does not get accepted answer function will return None

Listing 11 shows a sample of the function used to list cloned servers.

```

def list_cloned_servers_uuid():
    tag = open('cloned_servers_tags', 'r').read()
    servers = open("servers", "wb+")
    r = api_get('/tags/' + tag + '/servers')
    j = json.loads(r)
    if j:
        objs = j["objects"]

```

```

for o in objs:
    if not o['uuid'] == None:
        uuid = o['uuid']
        servers.write(uuid + "\n")
else:
    return None

```

Listing 11: A sample of the function used to list cloned servers.

Listing 11 shows how list_cloned_servers_uuid function works:

- tag gets compute instance uuid from cloned_servers_tags file.
- servers opens a file called “servers” to where cloned servers uuids are written.
- r uses api_get function to get cloned servers uuids.
- j gets API answer values in json format.
 - j is looped through and each server uuid is written in servers file.
- if j does not get accepted answer function will return as “None”.

Listing 12 shows a sample of the function used to stop cloned servers.

```

def stop_cloned_servers():
    with open('servers') as f:
        servers = f.read().splitlines()
    if servers:
        for uuid in servers:
            stop = api_post('/servers/' + uuid + '/action/?do=shutdown')
    else:
        return None

```

Listing 12: A sample of the function used to stop cloned servers.

Listing 12 shows how stop_cloned_servers_uuid function works:

- servers file is opened as f.
- servers file will be read without new lines.
- for each uuid api_post function is used to shutdown the server-client.

Listing 13 shows a sample of the function used to delete cloned servers.

```
def delete_cloned_servers():
    with open('servers') as f:
        servers = f.read().splitlines()
        if servers:
            for uuid in servers:
                delete = requests.delete(url + '/servers/' + uuid + '/?recurse=all_drives',
headers=header)
                if delete.status_code == 200 or 201 or 202 or 204:
                    return None
                else:
                    print delete.status_code('\n' 'custom message')
            else:
                return None
```

Listing 13: A sample of the function used to delete cloned servers.

Listing 13 shows how delete_cloned_servers function works:

- servers file is opened as f.
- servers file will be read without new lines.
- for each server Python requests function is used with delete payload to delete the server and servers hard-drives.
- if delete.status_code is 200, 201, 202, 204 function goes through.
- if delete.status_code is different than above it will be printed with custom message.

Listing 14 shows a sample of the function used to call cloudsigma API.

```

>def api_post(api_post, data=None):
    if not data:
>>     r = requests.post(url + api_post, headers=header)
    else:
>>     r = requests.post(url + api_post, data=data, headers=header)
    if r.status_code == 200 or 201 or 202 or 204:
        return r.content.strip()
    else:
        print r.status_code('\n' 'Something went wrong, please refer to wiki.')
        exit(1)

```

Listing 14: A sample of the function used to call cloudsigma API.

Listing 14 shows an example of API function which is used to send calls to cloudsigma API:

- if no extra data in input, then only normal call is issued.
- if extra data in input, then payload is added before headers. Payload can be example a server uuid.
- if return status code is correct content is stripped.
- If returns status code is bad return code is printed with informative text.

Functions listed above do provide all needed functions to fully automate provisioning, listing, stopping and destroying compute nodes. However if an destroying of an provisioned compute node fails, one needs to delete it manually. This can be done through CloudSigma GUI.

5.3 Ansible for Automated Tasks

One of the demands for the present project was to automate server image upgrades and FMI specific package uploads to CloudSigma servers. This was done with Ansible. Two Ansible playbooks were written:

- himan.push.new.packages.and.genrate.repometadata.yaml: for uploading FMI specific packages and to generate RPM-repository.

- update_cloudsigma.yml: update CentOS and FMI packages in the images, does a reboot if needed.

These Ansible playbooks are run by ecFlow when new FMI packages are available. If critical security patches are available Ansible playbooks are run by user manually.

Listing 15 shows the structure of the Ansible playbook which was used to upload FMI specific packages and generate RPM repository.

```
---
- hosts: cloudsigmadb
  sudo: false

  tasks:
    - name: push new fmi packages to himan
      copy: src={{ item }} dest=/var/www/html/RPMS/ owner=root group=root mode=0644
      with_fileglob:
        - /home/dev/fmi/ansible/rpms/*
      ignore_errors: true

    - name: update fmi.repo metadata.
      shell: createrepo --update -v /var/www/html/RPMS/
      ignore_errors: true
      register: metadata

    - name: send email
      local_action: mail
        host='smtp@foo.fi'
        port=25
        from="example@foo.fi (example)"
        to="example <example@foo.fi>"
        subject="cloudsigma Ansible report"
```

```
body="New packages are available, please run update_cloudsigma.yml
playbook"
ignore_errors: true
```

Listing 15: The structure of ansible-playbook which was used to upload FMI specific packages and generate RPM-repository.

Tasks used in Listing 15 are explained below:

- hosts: determines to which hosts playbook is pointed
- sudo: determines if a become method is used, in this case it is false and upload is done as root user.
- tasks: lists all “tasks” in the playbook.
- name: “task” name, after name comes module which has task to be done. In this playbook there are three “tasks” with following tasks:
 - copy: copies the packages to cloudsigmadb server
 - shell: generate new RPM-repository metadata
 - local_action: sends email that new packages were uploaded.

Ansible playbook presented above is just an example. It is good practice not to login straight as a root user, one should use “become” method to gain needed privileges, when using “become” method it is recommended to use ansible-vault to store private information. Ansible-vault encrypts private information and when running a playbook which uses ansible-vault, vault password is requested.

6. Testing

This chapter explains how the testing was done. It goes through the test plan, it also covers problems found in the testing phase and how they were fixed. It is important to think of the all possible factors when generating a test plan. With a proper test plan one can avoid problems in the later phases of production. When using off premise services such as IaaS one should go through possible scenarios which can affect critically to the company's production and services.

6.1 Creating Test Plan

The test plan was created together with the case company's production group. Creating the test plan with more than a few people is a good practice. The test plan covered the following phases:

- testing cloudsigma.py and CloudSigma API performance and reliability.
- testing CloudSigma scaling performance.
- possible provider SLA violations and reliability problems.
- testing processing with small data set.
- testing processing with 24 hour data set.
- documenting possible SLA violations.
- documenting possible problems in the IaaS environment.
- documenting how possible problems were fixed.

All testing results and findings were documented in the case company's JIRA, from where the documentation will be moved to WIKI in later time. The testing was started immediately after the test plan was created.

6.2 Testing CloudSigma API Performance and Reliability

An ecFlow suite was created to start and stop a basic virtual machine to see the performance of the CloudSigma API and to see if there were any reliability issues which could affect production. This phase was the longest testing period since it was important

to see the reliability of the CloudSigma API. `cloudsigma.py` Python script was used to do the API calls, this was also a test to see whether the `cloudsigma.py` worked properly. Within this test phase several problems were found:

- Provisioning of virtual-machine can vary quite much, depending of the load in the CloudSigma data center. Due to this starting and stopping timing had to be changed a bit.
- One big SLA violation happened. CloudSigma had 24 hours API outage due to a badly configured database-server.

6.3 Testing CloudSigma Scaling Performance

The next step was to determine how well CloudSigma IaaS performs when provisioning a larger number of virtual machines at the same time. The test showed that provisioning a larger number of virtual machines can affect the time the virtual machine is up and running.

When provisioning four or less virtual machines, the provisioning time + start up time was between 15-30 seconds before one could login to all the servers. When provisioning more than four virtual machines the provisioning time + start up time raised above 01 minute before one could login to all the servers.

From the findings of the scaling performance test it was decided that the provisioning should be done at least three to four minutes before ECMWF starts data stream to make sure that all the needed resources are available. This time frame gives a big enough “safe marginal” if CloudSigma IaaS has a bigger load at that time.

6.4 Testing Processing with Data Sets

Processing tests were done by experts from STU’s production unit. The production unit is responsible for data processing. On the first phase, sample data set was uploaded into the server to see that the environment can produce correct data sets. On the next phase a bigger data set was uploaded corresponding to a “real-life” situation. For this phase

servers needed to alter so they could handle the bigger load without problems. The following alterations were performed:

- set 12 vCPUs to cloudsigma-compute image.
- set 16384 MB of memory to cloudsigma-compute image.
- grow cloudsigmadb disks bigger so 24-hour data set can fit in.
 - processed data disks: 10 GB → 65 GB.
 - raw data disk: 10 GB → 230 GB.

6.5 SLA Violations

Several SLA violations took place during the test phases. Both of them were critical ones since they would have affected the production heavily. These problems were generated by CloudSigma API and billing. The root cause for the API malfunction was found by CloudSigma experts and it was fixed.

The second problem was that the CloudSigma billing API was not working properly. When one provisioned a virtual machine which had a private VLAN enabled and then deleted the whole virtual machine, an entry was left in to the database that user has these VLANs still in use. This problem generated a “snowball” effect. Provisioning of virtual machines is done several times per day and each time when provisioning happens virtual machines get a new unique ID which is entered in to the database, this caused that after each provisioning round one got charged from all the VLANs and ran out of credits very fast. In the worst case scenario the customer does not notice that the credits have ran out and all the resources will be deleted. CloudSigma experts did find the problems and fixed them.

7. Integration to Current Production Environment

This chapter covers how the CloudSigma post-processing environment was integrated to the current production environment. Before integration several steps were done to make sure that the post-processing environment was ready for production use:

- analysis of post-processing environment test results.
- add two firewall rules.

7.1 Analysis of Post-processing Environment Test Results

The analysis of the post-processing environment test results was needed to make sure that everything was working correctly and everything had been tested properly. The testing results did show several things which needed to be done before the integration.

Since both virtual machine images change when they are upgraded, rolling backups are needed. Backups have to be saved to the case company's disks which are then backed up into the tape library. This was done to make possible recovery faster and more reliable. Backups are done after each upgrade so if a recovery is needed virtual machine images will be on the latest version.

Account information had to be changed so all the emails pointed to the account would be received by all the members of the team responsible for the CloudSigma production environment. This is important since for example if the same billing problem occurred again, information about it would go to the correct people and correct actions could be taken immediately.

7.2 Changing Firewall Rules for Production Phase

To make the post-processing environment ready for production, two new firewall rules had to be added to cloudsigmadb. These rules allow ECMWF to stream data via FTP to

cloudsigmadb -server and allow FMI to download data via FTP from cloudsigmadb. FTP needs two ports to work TCP/21 and TCP/20. TCP/21 is for data traffic and TCP/20 is for connection tracking. One needs to enable the iptables module for connection tracking in “/etc/sysconfig/iptables-config” by adding the line presented in Listing 16.

```
IPTABLES_MODULES="ip_conntrack_ftp"
```

Listing 16: Syntax to allow FTP connection tracking.

TCP/21 for data traffic has to be added to iptables rules to allow FTP connections. Listing 17 show the syntax of iptables rule to allow FTP connection.

```
iptables -A INPUT -s 192.168.7.234/32 -i eth0 -p tcp -m tcp --dport 21 -j ACCEPT
```

Listing 17: Syntax of iptables rule to allow FTP connection.

Syntax shown in listing 17 allows connections from ip-address 192.168.7.234 with FTP protocol. After entering the following settings presented in listings 16 and 17, iptables rules need to be saved with command “iptables-save” and iptables need to be restarted so the connection tracking module gets loaded, the command to restart iptables is “systemctl restart iptables”.

7.3 Integration to Production Environment

The final step was to add the CloudSigma post-processing environment to the production ecFlow. This was done by STU’s production unit experts. A production CloudSigma ecFlow suite was created. CloudSigma ecFlow suite uses cloudsigma-Python script to issue calls to CloudSigma API. Cloudsigma ecFlow suite does the following tasks:

- start cloudsigmadb server
- provision N amount of clones from cloudsigma-image. N value comes from input, so it can be easily changed on the fly by operator.
- control workload for compute clones.

- stop and destroy cloudsigma-compute images.
- stop cloudsigmadb-server when processed data set has been transferred to FMI premises.

Yet again several test runs were done to make sure that integration was working properly. After the test were done successfully it was agreed that the CloudSigma post-processing environment was ready to be part of the daily weather production environment.

8. Discussion and Analysis

This chapter discusses the outcome of the project, it also analyses the obstacles faced and possible future steps in the case company.

The goal of this project was to design and build a post-processing environment to the CloudSigma IaaS environment. The project was done within 11 months, several delays were experienced but these were due to other projects which had an overlapping timetable. The goals of the project were reached, the post-processing environment was taken in as a part of the weather production infrastructure.

The present study provides necessary information as for what is needed to build a post-processing environment to an external IaaS. It must be understood that each cloud vendor's platform and API has its own features and limitations, also it must be understood that the technologies and solutions presented in the present study were chosen due to the requirements given to this particular project.

The post-processing environment design had a requirement that it had to be easily portable. This was due to the reason that it was the first time the case company was using an IaaS platform as a part of the production infrastructure and it is possible that other cloud providers will also be used as a part of the production in the future. As it was the first time this kind of a setup was built, a relatively large amount of time was spent on the project to research the best practices to design and build the post-processing environment. The portability goal was reached by having virtual machine images in RAW format, so they can be uploaded to any other cloud vendor's infrastructure which supports importing virtual machine images with a minimal effort. The testing migration to some other cloud vendors' IaaS environment were left outside the project, though it was discussed that in a later point it would be good practice to test migration to see how much work time the migration process needs.

Even though cloud services are marketed as "always available and safe" there is always a risk that the cloud service provider faces some serious issues and services are not available for an undefined time. There have been several cases when a bigger cloud

provider has had serious problems and they have affected their customers and their services.

During the project several serious problems were faced, which would have affected the production. From these problems it was understood that one should analyze all possible risk factors when starting to use off premise services as a part of the production. It is good practice to analyze whether it is safe to migrate a certain service to a cloud or whether it is better to run it in on premises. It was also understood how important it is to have everything backed up. A good practice is to make rolling backups for example when changes have been made, another important point is to test that backups are really working. From these backups it is easy and fast to rebuild the whole post-processing environment from scratch. With tools such as Ansible one can easily rebuild a post-processing environment with minimal effort.

9. References

1. interoute - **What is IaaS?** URL: <http://www.interoute.com/what-iaas> Accessed 30 March 2016
2. SoftwareInsider – **Compare Cloud Computing Providers.** URL: https://s3.graphiq.com/sites/default/files/252/media/images/_1570889.jpg Accessed 30 March 2016
3. libvirt.org – **Documentation.** URL: <https://libvirt.org/docs.html> Accessed 1 April 2016
4. oVirt.org – **Documentation.** URL: <https://www.ovirt.org/documentation/> Accessed 1 April 2016
5. ibm.com – **Network file systems and Linux.** URL: <https://www.ibm.com/developerworks/library/l-network-fileystems/> Accessed 1 April 2016
6. ietf.org – **NFS Version 3 Protocol Specification.** URL: <https://www.ietf.org/rfc/rfc1813.txt> Accessed 1 April 2016
7. python.org – **General Python FAQ.** URL: <https://docs.python.org/2/faq/general.html> Accessed 8 April 2016
8. python.org – **Python 2.7.11.** URL: <https://www.python.org/downloads/release/python-2711/> Accessed 1 April 2016
9. python.org – **Python 3.5.1.** URL: <https://www.python.org/downloads/release/python-351/> Accessed 1 April 2016
10. ietf.org - **Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content.** URL: <https://tools.ietf.org/html/rfc7231> Accessed 8 April 2016

11. cloudsigma-docs.readthedocs.org – **Welcome to CloudSigma API documentation!** URL: <https://cloudsigma-docs.readthedocs.org/en/2.14/> Accessed 16 Apr 2016
12. cloudsigma-docs.readthedocs.org – **Authentication.** URL: <https://cloudsigma-docs.readthedocs.org/en/2.14/general.html#authentication> Accessed 16 Apr 2016
13. software.ecmwf.int – **ECFLOW Introduction.** URL: <https://software.ecmwf.int/wiki/plugins/servlet/mobile#content/view/52464700> Accessed 16 Apr 2016
14. inspire.ec.europa.eu – **About INSPIRE.** URL: <http://inspire.ec.europa.eu/index.cfm/pageid/48> Accessed 16 Apr 2016
15. ansible.com – **Ansible Documentation.** URL: <https://docs.ansible.com/ansible> Accessed 9 Jan 2017
16. acces.redhat.com – **Introduction to the Red Hat SELinux Guide.** URL: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/SELinux_Guide/selg-preface-0011.html Accessed 16 Feb 2017
17. cloudsigma-docs.readthedocs.io - **Response Status Codes.** URL: <https://cloudsigma-docs.readthedocs.io/en/2.14/general.html#response-status-codes> Accessed 17 Feb 2017
18. ietf.org - **Hypertext Transfer Protocol -- HTTP/1.1** URL: <https://tools.ietf.org/html/rfc2616> Accessed 7 May 2017