FUTURE COMMUNICATION ARCHITECTURE FOR MOBILE CLOUD SERVICES

Acronym: Mobile Cloud Networking
Project No: 318109

Integrated Project
FP7-ICT-2011-8
Duration: 2012/11/01-2015/10/31

D2.5

Final Overall Architecture Definition, Release 2

Type	Public
Deliverable No:	D2.5
Work package:	WP2
Leading partner:	ZHAW
Author(s):	Andy Edmonds, Thomas Michael Bohnert (Editors), List of Authors overleaf
Dissemination level:	Public
Status:	Final
Date:	30 April 2015
Version:	0.5

List of Authors (in alphabetical order):

Thomas Michael Bohnert	ZHAW
Giuseppe Carella	TUB
Florian Dudouet	ZHAW
Andy Edmonds	ZHAW
Lúcio Ferreira	INOV
André Gomes	UBERN
Piyush Harsh	ZHAW
Gregory Katsaros	INTEL
Sina Khatibi	INOV
Andrea Marcarini	ITALTEL
Thijs Metsch	INTEL
Julius Müller	TUB
Navid Nikaein	EURECOM
Simone Ruffino	TI
Santiago Ruiz	STT
Giovanni Toffetti	ZHAW
Mohammad Valipoor	STT
Zhongliang Zhao	UBERN

Reviewers:

Giuseppe Carella (Peer)	TUB
Thijs Metsch (Peer)	INTEL
Lúcio Ferreira (GA)	INOV

Versioning and Contribution History

Version	Description	Contributors
0.1	First draft	Andy Edmonds et al.
0.2	Peer review version	Guiseppe Carella, Thijs Metsch.
0.3	Edited based on peer reviews	Andy Edmonds et al.
0.4	GA review version	Lucio Ferreira
0.5	Final version	Andy Edmonds et al.

Executive Summary

This document presents the updated version of the first MobileCloud Networking architecture, which was reported in D2.2. This architecture is one that is based on the requirements and scenarios, as set out in Deliverable 2.1 (D2.1, 2013) and positions itself as the core architecture to which all other technical work should adhere to in Mobile Cloud Networking. The key principles of Cloud Computing, as defined by NIST and Service Oriented Architecture still hold for the MCN architecture.

In order to reach to a stable architecture, implementations were provided. Based on the knowledge and experience acquired, a feedback process was employed to receive inputs for any updates required of the architecture. This detailed work is reported in the various technical work package deliverables, the most recent and up to date being D3.4, D4.4 and D5.4. However and importantly the experiences of each individual service owners are recounted and recommendations on how best to architect MCN services are presented.

This deliverable details any incremental change, the management interfaces that have been used throughout all the services developed and/or used in MCN. A technical reference implementation of the MCN architecture is presented and recommendations for architecting MCN services based on learnings is presented. Not only has the first architecture remained valid but so too has the mappings from it to the ETSI NFV architecture.

Summarily, there were no changes to the MCN lifecycle and the following logical architectural components still remain true to the first architecture:

- **Service Manager**: it provides an external interface to the EEU so that they can request the creation of supported services offered. A service provider (business domain) operates this and the SM manages all service orchestrators, which in their turn perform internal and offer external management of and to the EEU's service instance.

  The architecture of this component has not changed.
- **Service Orchestrator**: it oversees the end-to-end orchestration of a service instance and is a domain specific component, particular to the service provider. Each service instance is managed by a SO.

  The architecture of this component has changed with the addition of the Resolver component.
- **Cloud Controller**: It is responsible for supporting the service orchestrators requirements and complementing the service orchestrators' service life-cycle management needs in MobileCloud Networking. It provides the means to create and access services that higher level services need to build upon.

  The architecture of this component has not changed.

Table of Contents

List of Figures

List of Tables

List of Acronyms

3GPP	3rd Generation Partnership Project
AAA	Authentication, Authorisation, and Accounting
ABC	Always Best Connected
API	Application Programming Interface
ASP	Application Services Provider
BBU	Base Band Unit
BS	Base Station
BSM	Business Service Manager
BSS	Business Support System
CAPEX	Capital Expenditure
CDN	Content Delivery Network
CDR	Charging Data Record
CIP	Cloud Infrastructure Provider
COTS	Commercial Off-The-Shelf
CPP	Cloud Product Provider
CPU	Central Processing Unit
CRM	Customer Relationship Management
CSP	Cloud Service Provider
DC	Data Centre
DBaaS	
DIP	Data Centre Infrastructure Provider
DoW	Description of Work
DPI	Deep Packet Inspection
DS	Digital Signage
DSN	Digital Signage Network
E2E	End-to-End
EEU	Enterprise End User
eICIC	Enhanced Inter-cell Interference Coordination
EPC	Evolved Packet Core
EPCaaS	EPC-as-a-Service
EPS	Evolved Packet System

EU	End User
FP7	Framework Programme 7
GGSN	Gateway GPRS Support Node
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
HSS	Home Subscriber Server
HW	Hardware
IaaS	Infrastructure-as-a-Service
ICIC	Inter-Cell Interference Coordination
ICN	Information Centric Networking
IEU	Individual End User
IMS	IP Multimedia Subsystem
IMSaaS	IMS-as-a-Service
IP	Internet Protocol
ITG	
LAN	Local Area Network
LBaaS	
LCD	Liquid Crystal Display
LCR	Least Cost Routing
LTE	Long Term Evolution
M2M	Machine to Machine
MCN	Mobile Core Network
MCNC	(EPC) Mobile Core Network over a Cloud infrastructure
MCNP	Mobile Core Network Provider
MCNSP	Mobile Cloud Networking Service Provider
MME	Mobility Management Entity
MNO	Mobile Network Operator
MTC	Machine Type Communication
MVNO	Mobile Virtual Network Operator
NBI	North Bound Interface
NCP	Network Connectivity Provider
NIST	National Institute of Standards and Technology

O&M	Operation & Maintenance
OAM	Operation Administration and Management
OCCI	Open Cloud Computing Interface
OPEX	Operational Expenditure
OSS	Operation Support System
OTT	Over-The-Top
PaaS	Platform-as-a-Service
PCRF	Policy and Charging Rules Function
PDN	Packet Data Network
PGW	Packet Data Network Gateway
PoP	Point of Presence
QI	Query Interface
QoE	Quality of Experience
QoS	Quality of Service
RAN	Radio Access Network
RANaaS	RAN-as-a-Service
RANP	Radio Access Network Provider
RAT	Radio Access Technology
RC	Requirements Cluster
RE	Requirements Engineering
RRH	Remote Radio Head
RRU	Remote Radio Unit
RSS	Rich Site Summary
SaaS	Software-as-a-Service
SB	Service Broker
SDC	
SDN	Software-Defined Networking
SDO	Standards Development Organisation
SGSN	Serving GPRS Support Node
SGW	Signalling Gateway
SLA	Service Level Agreement
SM	Service Manager

SO	Service Orchestrator
SOA	Service Orientated Architectures
SSP	Support Systems Provider
STG	Service Template Graph
SW	Software
TD	Technical Domain
TSB	Technical Service Manager
UI	User Interface
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunication System
UP	Utility Provider
VAS	Value-Added Service
VM	Virtual Machine
VPNaaS	
VRAN	Virtual Radio Access Network
WP	Work Package
XaaS	Anything-as-a-Service

1 The MCN Architecture and Framework

Here we briefly describe the current MCN logical architecture, noting any particular changes since D2.2.

1.1 Key Motivations

Before detailing each logical architectural components, the key motivations and architectural principles of MCN are as follows.

The overall motivations for MCN and the architecture was introduced in D2.2. However the following goals MCN seeks to address are still valid. These are:

1. How can CAPEX/OPEX be optimised, offering the same service at a lower cost or with greater profit and,
2. How can existing subscriber bases be incentivised to use new and innovative services by efficiently leveraging the vast amounts of infrastructure at their disposal and in doing so create new revenue streams?

It was these very questions that MCN seeks to address. MCN is focused upon two key ideas as a means to address these challenges.

1. The first idea is to exploit cloud computing as infrastructure for future mobile network deployment and operation. At the core, MCN is about the movement from systems that are self-hosted, self-managed, self-maintained, on-premise designed and operated, to cloud-native-based software design that respects existing standards and management and operation of those designs instantiated as services or indeed, network function services.
2. The second idea is more visionary and forward looking. It envisions how future service providers endorse and adopt cloud computing services. These providers will leverage these services by building value-added services to commercially exploit as new innovative services, both traditional telecom services as well as new composed, end-to-end (E2E) services. These scenarios were documented in D2.1.

From these motivations, MCN naturally chose the NIST definition (Grance, 2011) of cloud computing and the principles of service oriented architecture. These principles have not changed and remain key stalwarts of the MCN architecture. Further information about them can be read in D2.2, section 2.1 and 2.3. Related to this are various definitions of terms used in MCN. These also have not needed updates and remain as previously agreed within the consortium. These can be referred to in D2.2, section 2.2.

1.2 MCN Service Management Framework

The MCN service management framework consists of two main areas of definition. The first is on the lifecycle used within the framework and the second is on the key MCN architectural logical entities. Both of these areas were defined in D2.2 and have not required any significant update based on implementation experiences and feedback.

1.2.1 MCN Service Lifecycle

MCN still maintains the lifecycle as defined in D2.2. This section merely provides a summary of what was detailed in that deliverable. In MCN, the service lifecycle has been divided into two distinct phases, the business phase and the technical phase. The MCN service lifecycle is related to the TMF service lifecycle and its mapping can be understood in D2.2. The first phase of the MCN lifecycle is the business phase, Figure 1, contains all activities related to the conceptualisation of the service, discovery and research of potential business partners that can offer services to be combined in the new service, the agreements of contracts between partners. This phase is largely a human- and manual-based process.

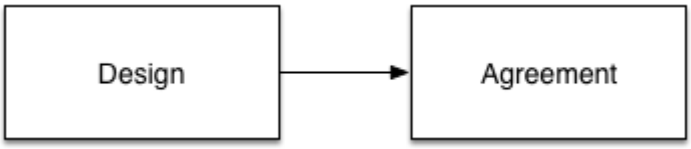


Figure 1 Business Phase Lifecycle

The second phase of the MCN lifecycle is the technical phase, as illustrated in Figure . It includes essentially all activities from technical design all the way through to technical disposal of a service. It is guided and governed by the business phase decisions and agreements between providers.

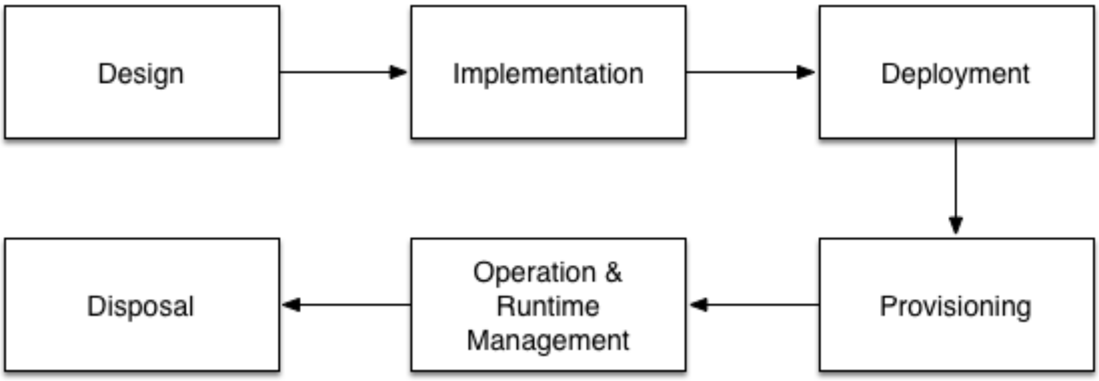


Figure 2 Technical Phase Lifecycle

At this phase, all aspects related to the business phase have taken place:

- **Design**: Design of the architecture, implementation, deployment, provisioning and operation solutions. Supports Service Owner to "design" their service
- **Implementation**: of the designed architecture, functions, interfaces, controllers, APIs, etc.
- **Deployment**: Deployment of the implemented elements, e.g. DCs, cloud, controllers, etc. Provide anything such that the service can be used, but don't provide access to the service.
- **Provisioning**: Provisioning of the service environment (e.g. NFs, interfaces, network, etc.). Activation of the service such that the user can actually use it. Examples:

- **Operation and Runtime Management**: in this stage the service instance is ready and running. Activities such as scaling, reconfiguration of Service Instance Components (SICs) are carried out here.
- **Disposal**: Release of SICs and the service instance itself is carried out here.

1.2.2 Key MCN Service Architectural Entities

Below is a very quick visual overview of the key entities, each of which being briefly summarised in the following sections. In the diagram of Figure 3 the blue entities are MCN key logical entities. Those in white are entities that are general and defined in D2.2.

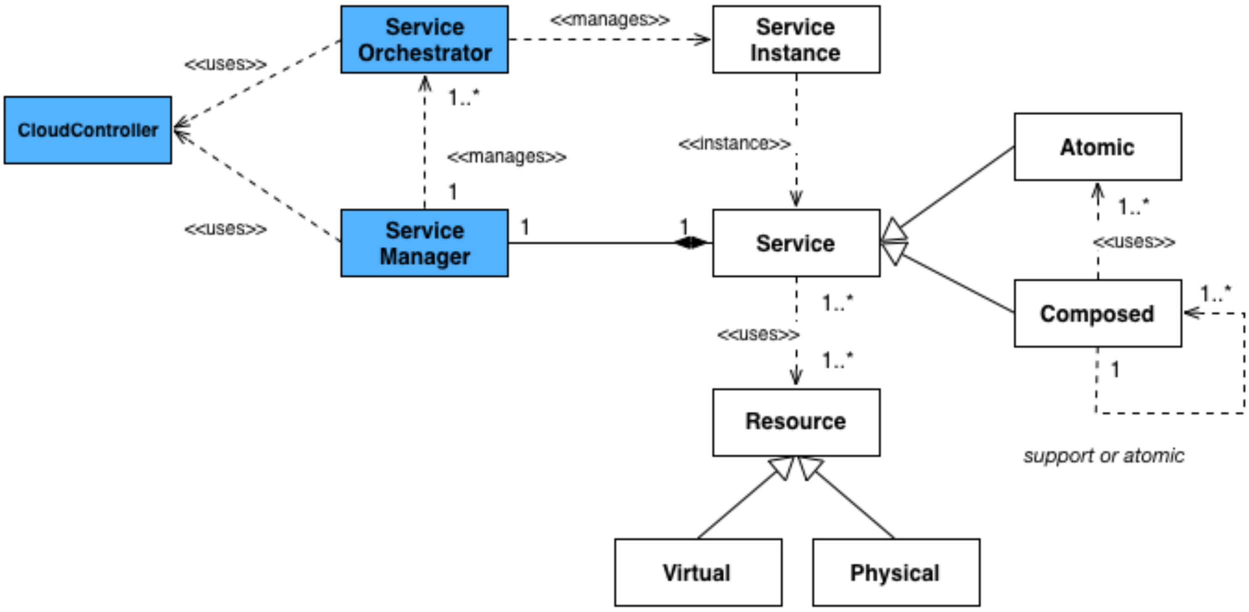


Figure 3 Key MCN Entities

1.2.2.1 Service Manager

The architecture of the Service Manager (SM) has not changed since D2.2. Nonetheless it has been heavily used by all services delivered out of MCN. For the all services in MCN, the implementation of the SM has been done such that there is minimal work for each service owner/developer, with a focus on the technical delivery of a the owner's service. Currently all that is required by owners and/or developers is to declare their service and what information that service offers to Enterprise End Users.

The service manager's architecture has satisfied all service owners in MCN but nonetheless, for completeness its architecture and a short summary of its functionalities and components are detailed below.

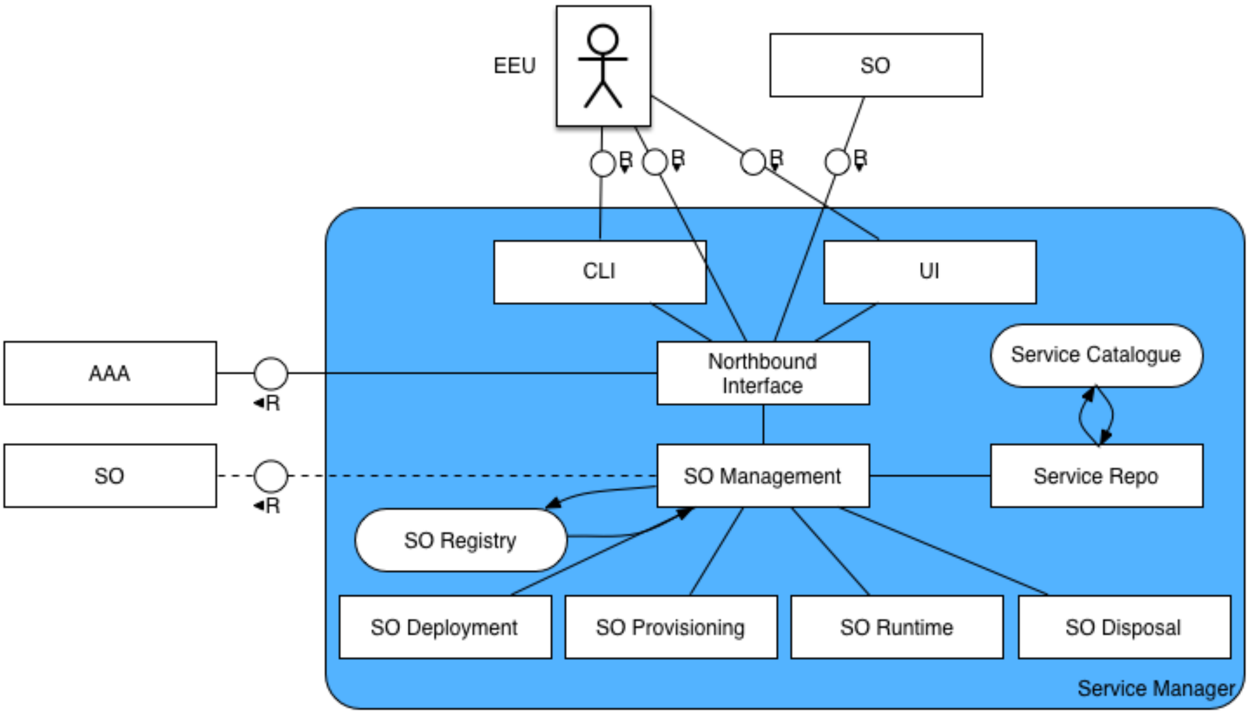


Figure 4 Service Manager

The SM provides an external interface to the EEU and is responsible for managing service orchestrators. It takes part in the Design, Deployment, Provisioning, Operation & Runtime Management and Disposal steps in the MCN Technical Lifecycle.

The SM's programmatic interface (northbound interface, NBI) is designed so it can provide either a CLI and/or a UI. Through the NBI, the SM gives the EEU or SO, both classed as tenant, capabilities to *create*, *list*, *detail*, *update* and *delete* (EEU) tenant service instance(s). The "Service Catalogue" contains a list of the available services offered by the provider. The "Service Repository" is the component that provides the functionality to access the "Service Catalogue". The "SO Management" (SOM) component has the task of receiving requests from the NBI and overseeing, initially, the deployment and provisioning of the service instance. Once the instantiation of a service is complete, the SOM component can oversee tasks related to runtime of the service instance and also disposal of the service instance. Service instances are tracked in the "SO Registry" component.

1.2.2.2 **Cloud Controller**

The current implementation of the Cloud Controller it based on the architecture described in D2.2 and D3.1. The architecture has not changed since then. For completeness its architecture is shown below.

f

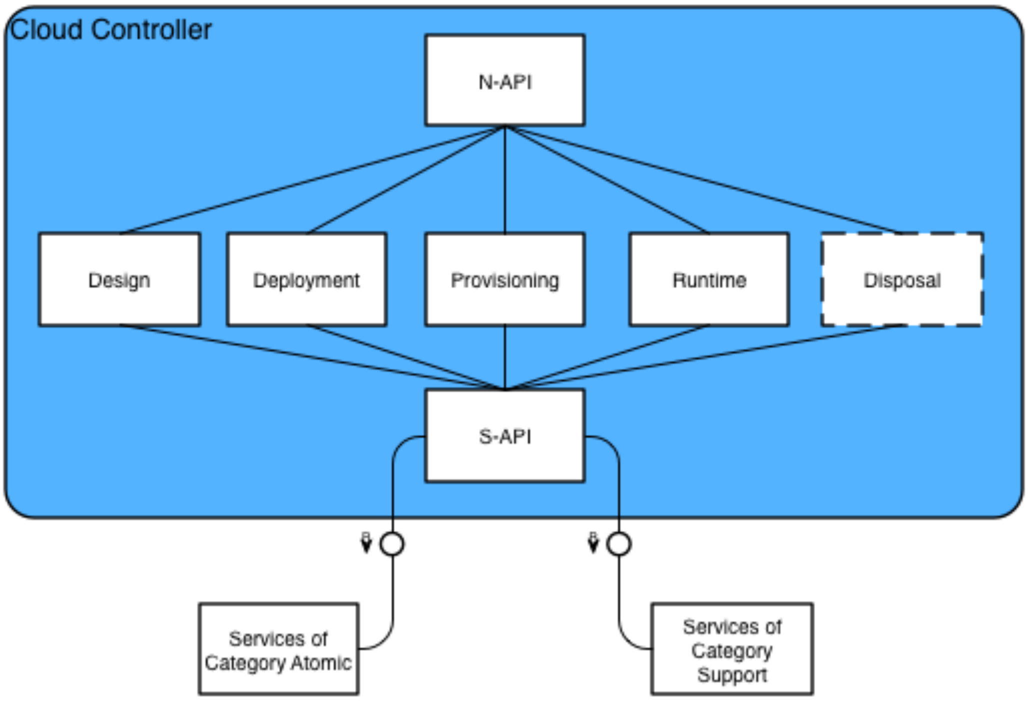


Figure 5 CloudController

The CloudController (full updated detail in D3.4) provides a key feature in the architecture of MCN. It abstracts from specific technologies that are used in MCN. Such examples are OpenStack Heat[1], Monasca[2] and Foreman[3]. This by having such a logical component and from the implementation perspective, should another provider that wishes to use another technology, they simply need to implement the relevant CloudController component.

1.2.2.3 Service Orchestrator

There has been minimal architectural changes to the service orchestrator over the implementation periods of WP3, 4 and 5. Where it has evolved has been in the understanding and consequent update of the software interface of the SO that a SO implementer must implement.

Another more significant change in the SO itself has been in the support of allowing SOs create external service instances in order to access functionality provided by those services. This functionality is provided by the SO's Resolver component, as depicted in Figure 6. The developer of the service does not need to implement anything, except to declare the required service dependencies. The declaration of these services are done in the service template graph and is technically realised by what is called a service manifest. Details of the resolver and the service manifest can be found in D3.4.

[1] https://wiki.openstack.org/wiki/Heat
[2] https://wiki.openstack.org/wiki/Monasca
[3] http://theforeman.org/

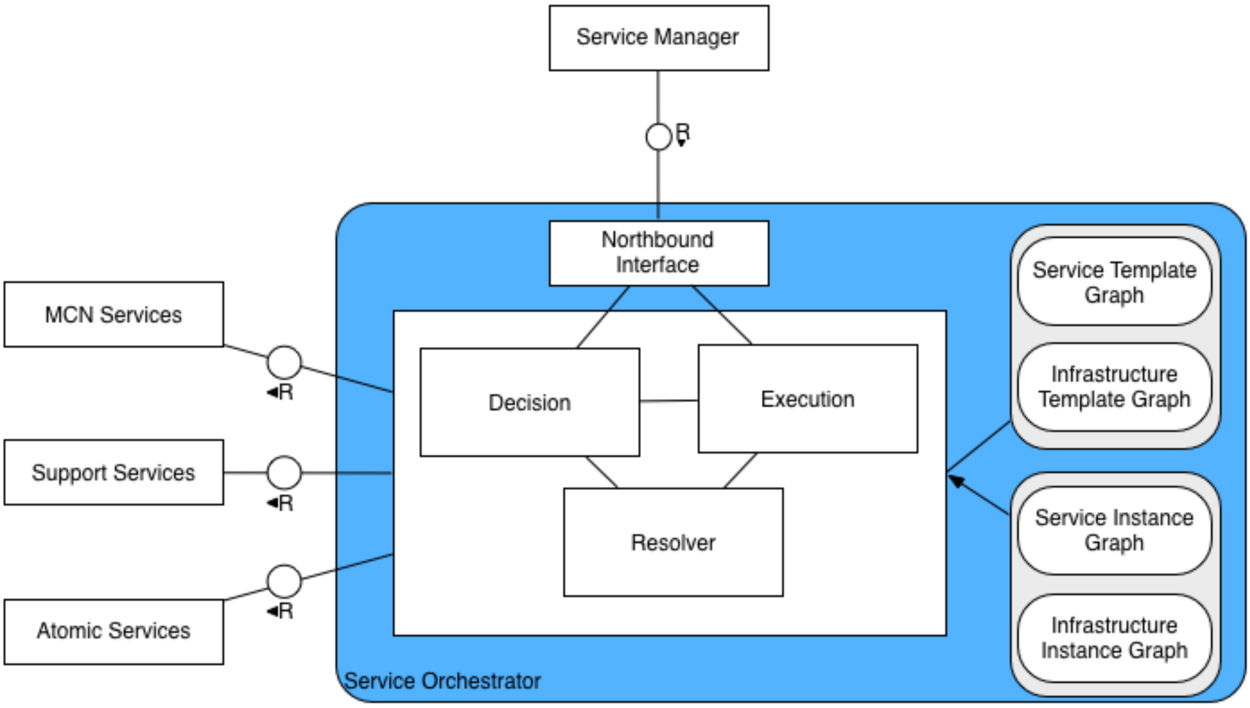


Figure 6 Service Orchestrator

1.2.2.4 Service Template Graph and Infrastructure Template Graph

These are two major concepts realised as templates, which play a role in the MCN framework. The concept of service and infrastructure template graphs has remained and remain the main way to logically describe the resources and services required by a service provider to offer their service. Both these concepts have been realised and implemented. The technical details of these can be found in D3.4. They are represented in the form of graphs with nodes and edges:

1. the Service Template Graph (STG) which defines how services can and should be composed together. For example the EPCaaS can have a requirement on the MaaS. This requirement is hence represented as a dependency. The STG interface can be queried through the Service Manager's NBI
2. the Infrastructure Template Graph (ITG) which defines how resources should be composed to be able to host Service Instance Components. For example the Analytics service requires two virtual machines: one to handle compute execution and one to handle the storage backend, both of which are connected through a network. Within the MCN framework ITGs are handled by templates documents which can placed upon different infrastructure service providers such as CloudSigma, Amazon EC2 as well as OpenStack and Joyent Smart Data Center. The Service Orchestrator can hold multiple ITGs, allowing for multi-region/zone deployments, in the SO bundle as well compute them on the fly.

Once a service instance is created the STG and ITG are instantiated. We define them then as:

1. The Service Instance Graph (SIG), which shows which service instance is connected to which other service instance. E.g. the EPCaaS instance '1' uses the MaaS instance '101'. This information is represented through the NBI of the SM.

2. The Infrastructure Instance Graph (IIG) which represents how resources instances are connected to each other. This include detailed information about the virtual resource instances - such as MAC & IP address, instance size and type definitions. This graph is handled within an SO instance.

1.2.2.5 **Composing End-to-End Services**

The MCN architecture has always had the basic requirement to be able to take multiple service types and be able to combine them and deliver them working as one "end-to-end" service. This requirement comes from MCN's adherence of the 'composable' service oriented architecture principle. According to this services may "*compose others, allowing logic to be represented at different levels of granularity. This allows for reusability and the creation of service abstraction layers and/or platforms.*"

It is the delivery of a service instance that can potentially span multiple administrative and business domains that we in MCN define as a End-to-End service instance. The ability to compose services in MCN is a functionality of the service orchestrator.

Where the MCN architecture differs from many orchestration frameworks is that it ensures the creation and management of not only the foundational virtual resources required to operate the target service logic but also the external service requirements. These dependencies are executed upon specifically by the SO's "Resolver" component.

In order to operate, the resolver component requires to understand the service dependencies a specific service requires. This information is part of the SO's bundle that is instantiated to manage the creation of an EEU requested service instance.

These requirements are logically described by the SO bundle's STG. The service template graph describes not only the service offered but the service types that the target service requires to operate (for example, the DSS service *requires* the CDN service to operate).

In the context of a SO creating a service the following order of resource and service instance creation is followed:

1. Dependencies defined in both the ITG and the STG are deployed. This is possible as at this stage only the dependencies are to be created. The agreed rule here is that all services developed in MCN must be able to be independently deployed and then at a later stage provisioned with parameters. This separation is given by the MCN lifecycle and proves to be very useful by maximising parallelisation of the deployment phase.
2. Once all dependencies of the ITG and STG have been deployed, the STG dependencies are provisioned. In this case, the required parameters of the respective service dependency is supplied by the resolver component. The result of this is a graph of STG dependencies that are fully configured and readied for use by the ITG dependencies. It is the separation of deployment and provisioning phases that is a large advantage in also allowing for simple and effective configuration (provisioning phase).
3. Now with all STG dependencies deployed and provisioned the ITG dependencies of the service instance can now be configured with the endpoints of the external STG dependencies. Once this is complete, the service instance is then ready to provide service.

The logical sequence of the deployment and provisioning of a composed service instances is shown below. The implementation of providing composed service instance is further detailed in D3.4.

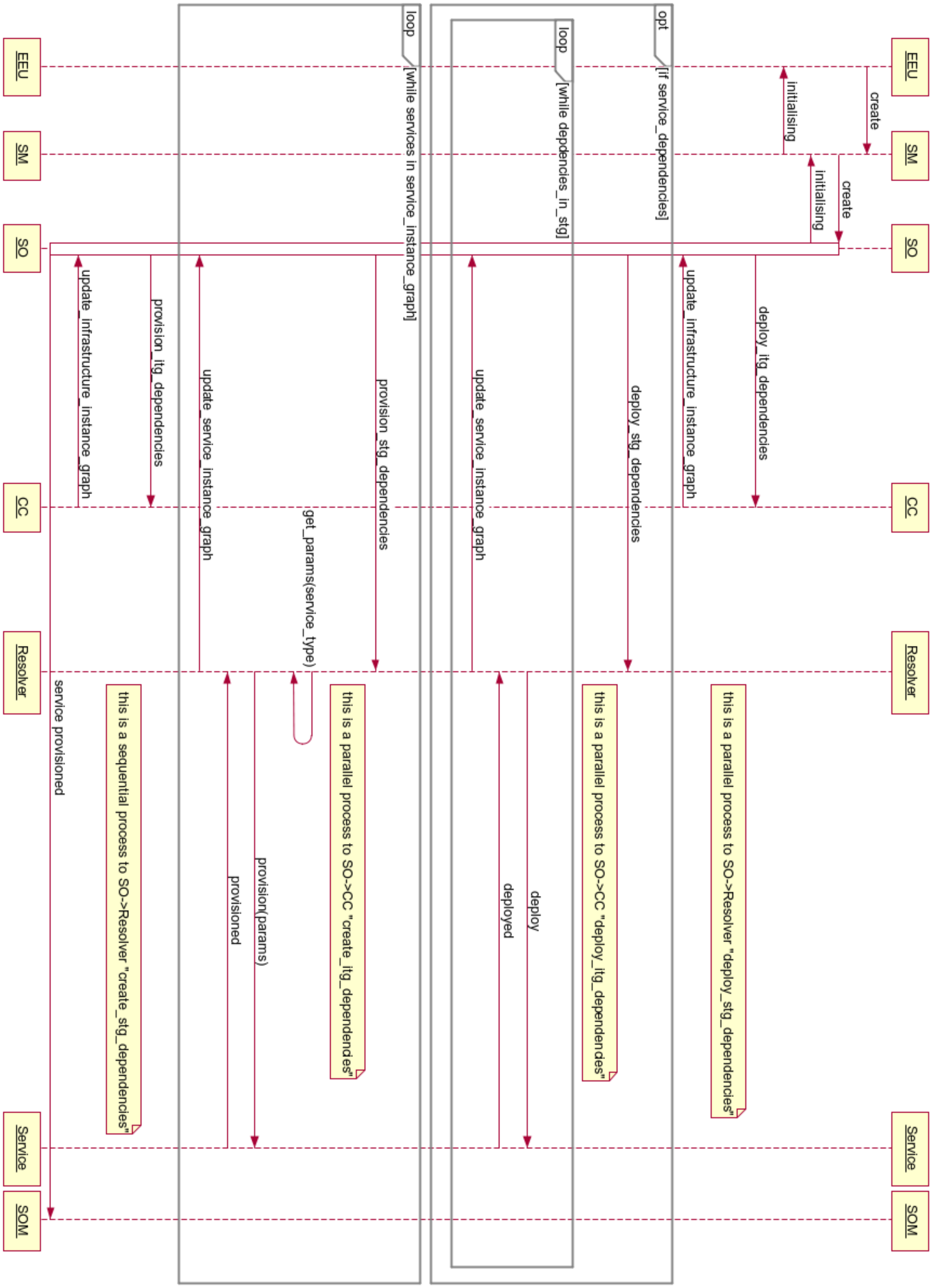

EEU
SM
SO
CC
Resolver
Service
SOM
create
initialising
create
initialising
opt
[if service_dependencies]
deploy_itg_dependencies
update_infrastructure_instance_graph
this is a parallel process to SO->Resolver "deploy_stg_dependencies"
loop
[while depdencies_in_stg]
deploy_stg_dependencies
this is a parallel process to SO->CC "deploy_itg_dependencies"
deploy
deployed
update_service_instance_graph
loop
[while services in service_instance_graph]
provision_stg_dependencies
this is a parallel process to SO->CC "create_itg_dependencies"
get_params(service_type)
provision(params)
provisioned
update_service_instance_graph
provision_itg_dependencies
update_infrastructure_instance_graph
this is a sequential process to SO->Resolver "create_stg_dependencies"
service provisioned

Figure 7 Sequence of Service Composition

2 MCN Management Interfaces

2.1 Service Common Interfaces

The MCN reference implementation was faced with a challenge early. This was given the number of services, how to maintain interoperability and a common interface and model between the difference services. To solve this, the Open Cloud Computing Interface (OCCI)[4] was selected as it had specified all the the basic needs of both the SM and SO northbound APIs in both its core model specification. OCCI also contains a common means to render that model and transfer it over a RESTful interface.

OCCI comprises a set of open, specifications documents delivered through the Open Grid Forum (OGF). The OCCI working group is developing the specification around the ideas of integration, innovation, portability, and, at the core, interoperability. OCCI's modular approach allows for extensibility, flexibility, and the discovery of capabilities. In order not to repeat the whole model here, this (Richardson, 2012) is a very useful, short and complete reference to OCCI.

The OCCI specification is setup like a toolbox allowing you to pick the parts you need and leaving out the parts which are unneeded. One central part is the core model which all types of OCCI compatible interface must realize. The core model is a meta-model which can be extended for particular uses cases. For example for modelling orchestration and management interface such as those used in MCN. By extending the core model for each individual service (through the help of the service manager library (D3.4)) services of arbitrary description can be served to EEUs.

The core model of OCCI defines a type model for linked resources. This model can then be rendered using an RESTful HTTP based interface. This is defined by the OCCI protocol and rendering specifications. The MCN project has greatly contributed to the development (D7.2.2) of OCCI revision 1.2.

Thanks to the extensibility feature of OCCI we had to define some base type for our Orchestration and Management interface. Since we have been working with the concept of STGs and ITGs - in form of graphs - the linked resources model of OCCI plays well with what we are trying to achieve.

Since an OCCI compatible interface can be queried to understand the capabilities of a resource, and those capabilities can be overloaded with the concept of Mixins, we get a scalable orchestration framework. So the base type for orchestrators is generic for all Service Orchestrator Instances in MCN, specific parts to the service could however still be model thanks to the Mixin capabilities. This means that the basic operations to trigger the life cycle of a service are defined in the orchestrator type while specific operations to a service are defined by an overloading mixin (deploy, provision, destroy vs set_dns_endpoint).

2.2 Service Specific Management Interfaces

The common OCCI interface to all service managers only provides the means to create, delete, update and get details of the service instances that have been provisioned for an Enterprise End User. For the specific service this is not enough however as with this alone no service-specific interface can be exposed. These service specific interfaces are provided in the details of a service instance and are

[4] http://www.occi-wg.org

typically RESTful based interfaces but specific to the service. These service specific interfaces are detailed in the services' relevant deliverable. The common interface specification and that of the specific interface for the relevant service can be found as part of D6.4.

3 Updating the MCN Architecture

In order to update the MCN architecture bi-weekly calls were held with owners of service implementing the MCN architecture. These inputs were first compiled and then another set of discussions were held to first see if:

1. the logical architecture already covered the comment,
2. if the comment was a technical comment more so directed at the MCN reference implementation of the architecture

If the comment did not fit these criteria, then the comment was accepted to be dealt with as an architectural update. Regardless of how each input received, a decision on how it was dealt with was agreed upon and recorded. Below is a table that lists the rationalized inputs received from service implementers and what the agreed decision was made regarding it.

Table 1 Architecture Inputs for Update

ID	Summary	Description	Decision
1	Scaling Services and SICs	Here clarification was sought on how should scaling be approached and if scaling cloud be automatically done.	Ability for scaling was already supported in the architecture of the Service Orchestrator. Service specific
2	Monitoring	How can monitoring be provided for services instances and their SOs that needed to be monitored.	This was specific to the reference implementation and the means to access a monitoring service was integrated into the MCN SDK.
3	Registration of services	A means to register service manager endpoints was required.	This was specific to the reference implementation and implemented accordingly. No update to the logical architecture was required.
4	Infrastructure graph management	This item dealt with how to get the latest version of the ITG as deployed through the CloudController so that it could be inspected and updated by the respective SO.This is related to #8	This was a question of the reference implementation. Here the solution was to use the MCN SDK to get and update the current ITG.
5	Orchestrator topologies	Here the item to address was if there should be an orchestrator per service or a common orchestrator.	This aspect was already answered in the first architectural deliverable, D2.2, section 3.5
6	Service Graphs	A way was needed to represent dependencies in the service graph.	This was an aspect of how the STG and ITG were implemented in the MCN reference implementation. It was realised through both the service manifest, heat templates. The STG was realised as a set of interlinked OCCI resources. Details of these solutions are in D3.4.
7	What technologies can be used and shared by all?	There are a number of technical functions that individual services/SOs share. How can these functions be shared?	This was related to the MCN reference implementation. The solution to this was to introduce the MCN service development kit (SDK), which contains a set of functions that can be shared. The SDK also allows service developers

			easily create an instance of a required service.
8	Scaling there an impact/conflict possible with the SLA service?	The consideration here was if a service instance was bound by a SLA, should it be possible for the service instance to be manually scaled by the EEU. This is related to #10.	This item was one related to the MCN reference implementation and in particular the MCN SLA service and the service which is guaranteed by the SLAaaS. As such it was considered service specific and did not require an update of the logical architecture.
9	Scaling manually done by EEU	The request was a question if the architecture could support EEU requests to scale their service instance.	This is possible as each service manager implementation is specific to the provider, who can decide to offer such capabilities out of the service manager interface. As such there was no requirement on the MCN logical architecture to change. Technically the support to enable this was added.
10	Revision of MCN Lifecycle	A build process is missing from the lifecycle. Including this will add notions of continuous build and integration into MCN, aiding agility fast deployment.	Where as it was agreed that this step in the technical phase could be added it was not pursued and was given low priority over other technical work.
11	Detail how and where OCCI is used	OCCI is used extensively in the MCN reference implemented. It should be explained where OCCI is used.	This was a comment against the reference implementation of the MCN architecture. However, it was agreed that it should form part of the architectural deliverable (D2.5).
12	Service state model	The service state model should be detailed to include the following states: ● initialise ● activate ● deploy ● provision ● active (entered into runtime ops) ● update ● destroying ● failed	This was deemed to be specific to the technical implementation and so no update to the logical architecture was required, given that the lifecycle described in the architecture covered these states. The states listed here have been implemented as part of the MCN reference implementation.
13	Topic of microservices should be dealt with and placed in the context of telco/NFV	Given the existing relationship of MCN to service oriented architecture, that relationship should be explained in context to the current development of microservices	This was accepted and is explained in this deliverable (D2.5).

4 Recommendations for Architecting MCN Services

In order to function optimally in a cloud computing environment, MCN services need to be carefully architected for it. In fact, running applications in the cloud efficiently requires much more than deploying software in virtual machines, it needs continuous management.

This section will discuss in which way the MCN architecture supports the task of architecting MCN services as cloud-native applications. Moreover, we will discuss the common pitfalls of cloud design as well as recommended architectural best practices.

4.1 Cloud-native applications

The term **cloud-native** is often used to indicate an application/service that:

- is optimized to run in the cloud (IaaS or PaaS), i.e., it is designed to be elastic (avoiding over and under-provisioning and resilient)
- takes full advantage of the cloud environment
- considers the drawbacks of the cloud environment

There are several advantages in embracing the cloud, but in essence they typically fall in two categories: either **operational** (flexibility / speed) or **economical** (costs) reasons.

From the former perspective, cloud computing offers fast self-service provisioning and task automation through APIs which allow to deploy and remove resources instantly, reduce wait time for provisioning dev/test/production environments, enabling improved agility and time-to-market facing business changes. Bottom line: **increased productivity**.

From the economical perspective, the **pay-per-use** model means that no upfront investment is needed for acquiring IT resources or for maintaining them, companies pay only for effectively used resources. Moreover, handing off the responsibility of maintaining physical IT infrastructure, companies can avoid capex in favor of opex and can focus on development rather than operations support.

However, even after a set of architectural patterns and best practices for cloud application development have been distilled e.g., (Wilder, 2012), (Fehling et al., 2014) (Homer et al., 2014), the challenge of bringing an existing application to the cloud (i.e., making it “cloud enabled”) or developing a “cloud-native” application from scratch is still very relevant in cloud development.

On one hand, a pay-per-use model only brings cost savings with respect to a dedicated (statically sized) system solution if 1) an application has varying load over time and 2) the application provider is able to allocate the “right”' amount of resources to it, avoiding both over-provisioning (paying for unneeded resources) and under-provisioning resulting in QoS degradation.

On the other hand, years of cloud development experience have taught practitioners that commodity hardware and switches often do break. Failure domains help isolate problems, but one should “plan for failure”, striving to produce resilient applications on partially reliable infrastructure.

More in detail, the main factors that contribute to failures and outages in cloud services are:

- **Unreliable infrastructure:** cloud computing infrastructures are based on commodity hardware. Failures in large commodity data centers are the norm rather than the exception (Bonvin et al, 2010), (Cockcroft. 2012) (Gill et al. 2011).

- **Unreliable third party services:** many services are obtained as the composition of atomic services possibly from different third party providers. A failure or a slow-down in one atomic service can potentially hinder the functioning of the whole composition (Filieri et al., 2012).
- **Varying load:** running conditions of services typically vary in time in terms of load intensity, workload mix, position of the users. For instance, public facing services are exposed to an external load which is not controllable and varies (sometimes extremely e.g., in the so-called slashdot effect) over time. Exposed to highly varying load, most simple services will break and become completely unavailable in the worst case, or in the best their response times will become so high to render the service practically unusable for its purpose. In other terms their quality of service (QoS) and experience (QoE) will degrade beyond repair.

Unreliable infrastructure in cloud computing means an application provider cannot expect any of the services from the cloud providers to be immune from failures: issues may span from hardware malfunctions in single hosts resulting in virtual machines disappearing, all the way to availability zone (AZ), or data center wide compute, networking, storage, or power failures resulting in entire AZs or data center services becoming unavailable.

Third party services delivering inconsistent QoS results in service compositions with highly variable response times and require discovering and utilizing multiple service instances and providers to mitigate risks.

Finally, unpredictable spikes in load require reactive management and admission control to prevent rendering a service inoperative.

Considering these aspects, providing reliability for services amounts to being able to face uncertainty and react to the changing conditions of the service, be them infrastructural, depending on third party services, or load.

Given the nature of the issues and the need for immediate reaction, automated management functionalities dealing with the above and other issues are the commonly adopted solutions in cloud development practice. In other words, management functionalities are essential to achieve reliable ICT services in the cloud.

The MCN architecture takes these requirements into account and provides hooks for automated management, in particular with respect to scalability, reliability, and placement management functionalities. The following subsections will deal with each of these aspects separately.

4.2 Cloud-native architectural principles

Drawing on service-oriented principles, cloud native applications are generally based on loosely coupled architectures and leverage asynchronous, non-blocking communication patterns. Key to a successful cloud-native design is accommodating features like resource pooling, multi tenancy, on demand and self-provisioning, and prominently scaling behaviour. The latter for instance, is based on monitoring target metrics as resources are added or removed. Against these metrics, logic runs such that when demand increases new resources are automatically added based on proactive and/or reactive actions. This is known as scaling out. The inverse applies too; when demand reduces resources are removed and is known as scaling-in. This scaling behaviour bring further reliability into the target. Scaling behaviour can deal with events so the target has no perceived downtime or quality of experience degradation, including transient failures. Inline with this, a cloud-native target should exhibit upgrades with no perceived downtime. As scaling actions are mainly part of the runtime phase of a target, it also presents the opportunity to carry out optimisations such as optimising for cost by reducing the number of resources when not necessary or placement of resources in the best geographical area to minimize latency. A recent embodiment of the above principles is in the microservices architectural style.

Applying cloud-native architectural principles to MCN services can be very challenging in practice. In fact, some of the assumptions of generic cloud computing (e.g., virtually unlimited resources, virtualization and abstraction from physical resources) are not valid in the context of some MCN services which, by providing network function virtualization are tightly bound to specific physical resources and geographic locations (e.g., consider for instance RANaaS).

4.2.1 Microservices

The "microservices"[5] architectural style has recently received considerable attention as a viable solution to building Internet scale applications by using composition of small, simple, isolated single-functionality services (i.e., microservices) into coherent applications. The MCN architecture was designed before these principles became best practices.

Microservices architecture is in opposition to the so-called "monolithic" architecture where all functionality is offered by a single logical executable.

[5] Martin Fowler. http://martinfowler.com/articles/microservices.html

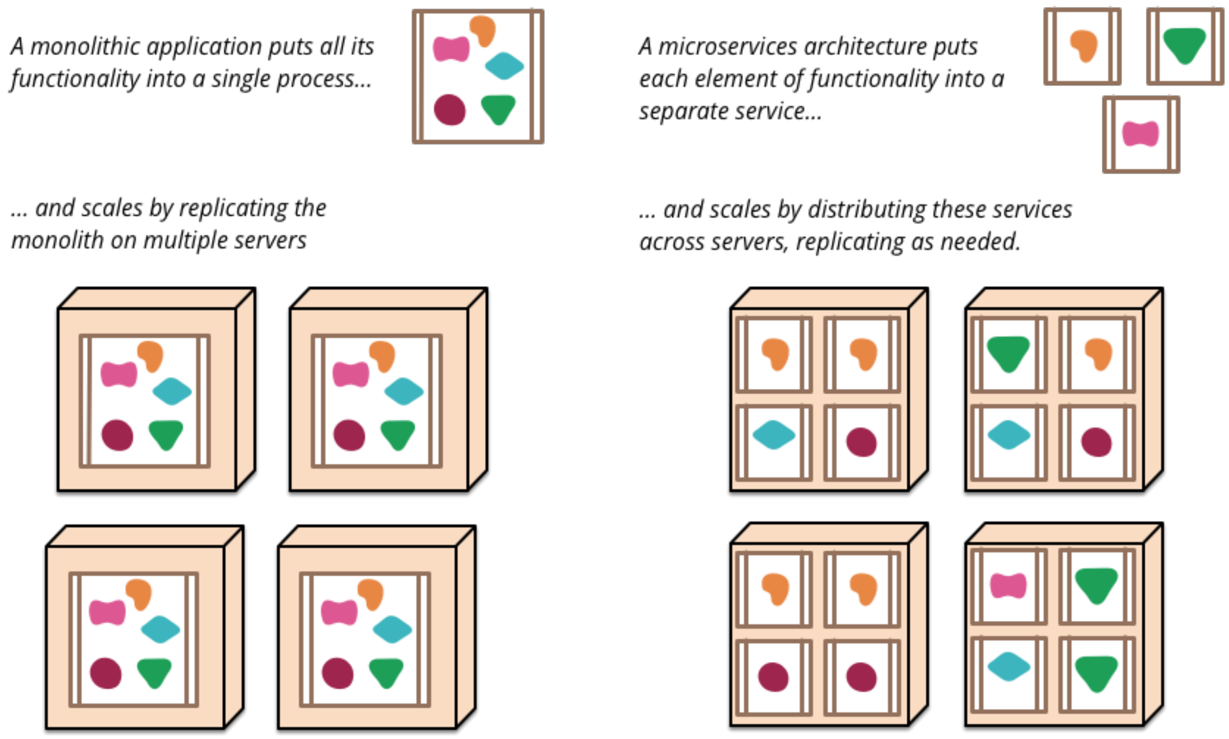

A monolithic application puts all its functionality into a single process...
A microservices architecture puts each element of functionality into a separate service...
... and scales by replicating the monolith on multiple servers
... and scales by distributing these services across servers, replicating as needed.

Figure 8 Microservice Scaling[6]

The principles behind microservices are:

- Loosely coupled services with clear boundaries defined by interfaces
- Microservice **independence**:
  - performance and failure isolation
  - delegation to a single team
  - own release cycle
  - best technology for the task
  - decentralized data management
- Infrastructural automation
- Design for failure

While many of the principles stem mostly from the requirement of making the continuous development and deployment process scalable in terms of involved teams, reducing dependencies and need for inter-team coordination, strong technical advantages of this architectural style are failure and performance isolation so that applications can (partially) continue to function even while some functionalities are not available.

[6] Image credits: http://martinfowler.com/articles/microservices.html

The main drawback of the microservices architectural style is that applications become more complex with several moving parts that need to be monitored and managed, so that extensive automation needs to be in place.

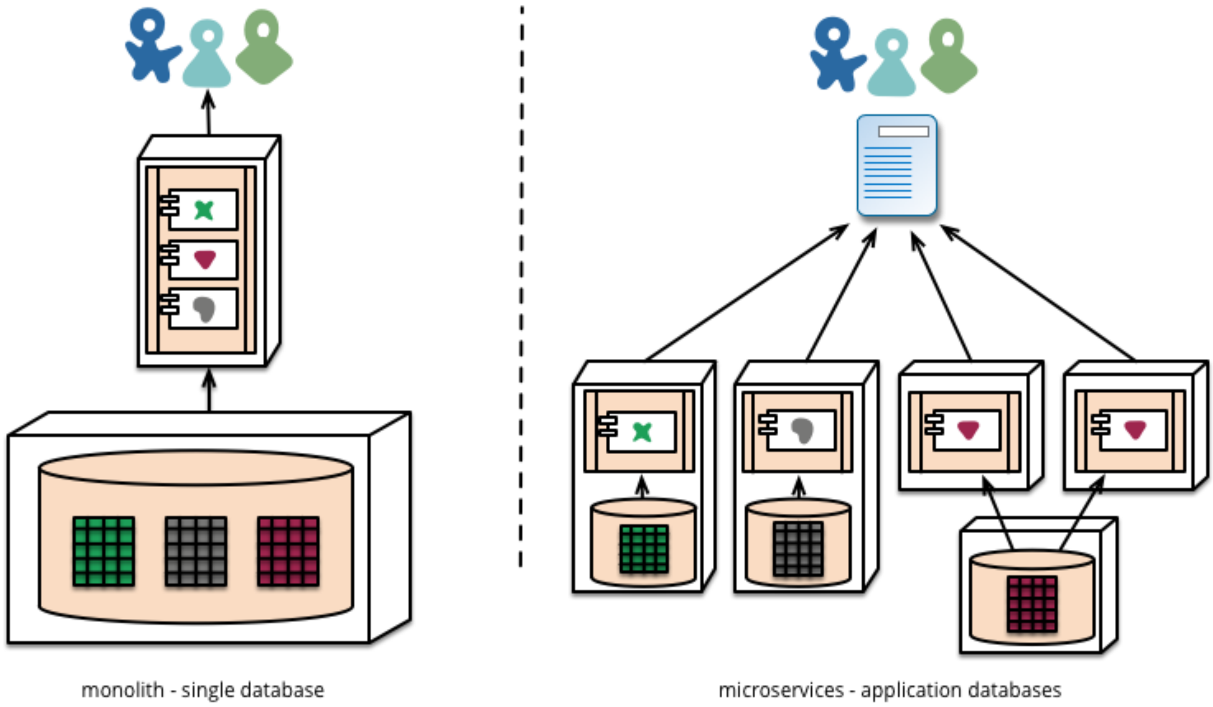

monolith - single database
microservices - application databases

Figure 9 Microservice Hetrogenity[7]

Microservices can use different development languages and database technologies in building a coherent modular service composition. Microservices are typically provided and managed by a single application provider, but it's easy to see how the same management principles needed to deal with (third party) service compositions need also to be applied “internally” to microservice compositions for the same overall service reliability goal.

In microservice architectures, several patterns can be used to guarantee resilient, fail-fast behavior. For instance, the circuit-breaker pattern (Nygard, 2007) or client-side load balancing such as in the Netflix Ribbon library[8]. The typical deployment has multiple instances of the same microservice running at the same time, possibly with underlying data synchronization mechanisms for stateful services. The rationale behind this choice is to be able to deploy microservice instances across data centers and infrastructure server providers and letting each microservice quickly adjust to failures by providing alternative endpoints for each service type.

4.2.2 Scalability

In the MCN architecture, scaling decisions are taken by the SO Decision module.

[7] Image credits: http://martinfowler.com/articles/microservices.html

[8] http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html

The architecture leaves total freedom to the MCN service developer to decide how to implement the scaling decision logic.

The common approach used by several MCN services has been to leverage the Monitoring as a Service component (MaaS) and have the SO Decision module poll monitoring data periodically in order to take scaling decisions. Actual scaling actions are taken by the SO Execution module which typically uses the CC to deploy or update a Heat template in OpenStack (e.g., see Figure 25 in D5.2). Indeed the key patterns in the MCN architecture are: 1) runtime module, 2) SO decision and 3) analytics (AaaS) triggering changes, however not in that order or combination. However the combination of the 3 is powerful.

In the last release, the MCN SDK has been extended to include and provide a reference implementation of the event detection and notification logic triggering auto-scaling decisions.

The implementation is based on Monasca, an "open-source multi-tenant, highly scalable, performant, fault-tolerant monitoring-as-a-service solution[9] that integrates with OpenStack". Details of this is available in D3.4.

The third approach include the usage of the Analytics service to do in depth analysis of service performance and actuate based on the outcomes of that analysis.

Appropriate scaling actions are service-specific and depend on the nature of the service and its components. Combining or picking one ot the 3 above offered approaches demonstrate how the MCN architecture allows for elastic on-demand usage of services.

For instance, stateless SICs can be deployed and disposed of without considering their internal state. Stateful components will require appropriate actions for state initialization, replication, reconciliation and migration.

In MCN, many services deal with network functions and are closely tied with physical infrastructure and communication flows. For instance, a service can include stateful data path entities requiring a fixed TCP connection between endpoints with real-time data exchange.

In this case scaling decisions imply reconfiguration actions to be coordinated in an atomic fashion to avoid (or minimize) service disruptions.

4.2.3 **Reliability**

The reliability of a service depends on its ability of being resilient to failures and environmental changes (e.g., in terms of load).

Scaling a service and setting admission control policies are also fundamental aspects of catering for service reliability. The previous section dealt in particular with adapting a service to changing load to avoid under-provisioning (and inherent QoS degradation all the way to non-availability) as well as over-provisioning (wasting resources). Here we discuss how to provide a reliable service on top of unreliable infrastructure and third party services.

The classic approach to mitigate failures is **redundancy**. In this aspect, cloud computing is novel and economically more viable with respect to traditional enterprise-grade systems in that it relies on

[9] https://wiki.openstack.org/wiki/Monasca

software automation (replicating and restarting software components) rather than (more) expensive hardware redundancy to provide resilience and availability on top of commodity hardware.

Redundancy against infrastructural failures boils down to replicating service components across different failure domains to achieve the desired level of availability. Spreading SICs respectively across different physical hosts, availability zones, data centers, and geographical regions results in lower probability of concurrent failures and higher availability.

Load balancing techniques and cloud-native architectural patterns (e.g., circuit breaker) can be used to distribute the load across components hiding failures from end users.

The same considerations we made in the previous section for stateless with respect to stateful service components or components depending on physical infrastructure apply here.

In particular, stateful components will require state replication mechanisms while service components tightly bound to physical devices or a specific geographic location will need to rely on some sort of physical redundancy.

Redundancy at service composition level can be achieved by orchestrating several service instances of the same service type in case of MCN services, or relying on more than one instance of functionally equivalent third party services.

4.3 Placement Based on Resource Requirements

The general problem of service placement and routing in cloud-native applications deals with planning and deciding what kind of resources to deploy, using which cloud provider and data center in order to maximize revenue providing the required QoS to the application's users.

This optimization can be seen as an instance of the stochastic bin packing problem (Shabtai et al., 2015.). Cloud resources have different performance, start-up and minimum billing life-time (e.g., one hour for Amazon EC2 on demand VMs) impacting on their costs per unit of time. In general longer-lived resources (e.g., reserved instances in Amazon EC2) have a lower cost per time unit than on demand resources.

The application provider wants to minimize the total cost of running his application subjected to a variable load coming from different locations over a given time horizon. Routing of traffic from different locations to specific data centers is also part of the control logic.

In the context of MCN, different considerations with respect to the general service placement problem apply.

As we also mentioned in the previous sections, some MCN services provide networking functionality and are tightly bound to the infrastructure they manage and use.

For these services (e.g., RANaaS) optimizing placement might not really be an option since the service itself requires to run at the specific geographic location requested by the EEU.

Other services involving both networking and compute components are more amenable to placement optimization, in particular with respect to the compute or storage part.

Placement optimization can be addressed before deploying a service that does not change its location (static placement), or while the service is running to optimize the placement according to the (changing) location of the end users. The latter approach is called dynamic placement and the logic controlling it is typically service-specific.

For instance, the Follow-Me-Cloud (FMC) algorithm in ICNaaS is responsible to take decision on the content to migrate according to the user movement and their interests. As such, the FMC is responsible for placing the information close to the point where the user is connected, or where the user will be connected in a short future. The underlying concept of FMC can be applied to allow the placement of services in MCN, by considering criteria such as current of future user location and user preferences.

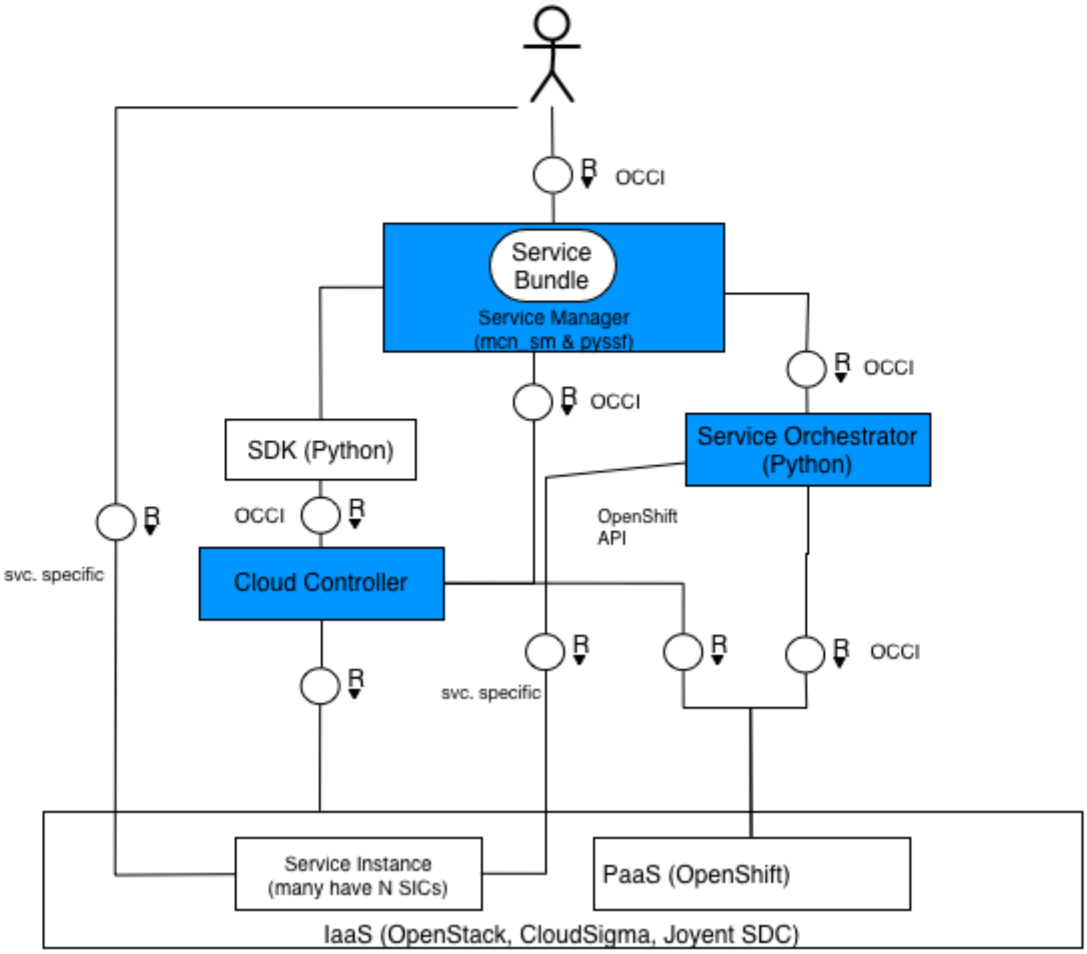
For instance, if the user is close to testbed A, a specific service, using the MCN architecture, can provide its functionalities relying on this information. Moreover, the follow-me-cloud has associated mobility predictions, targeting to estimate the user location in a certain future. Based on this location services can be placed close to the location of users.

Summing up this recommendation section, we can state that albeit the MCN architecture and services were designed before the microservices design pattern came to be, MCN architecture and services still follow most of the ground principles of microservice architectures and cloud-native applications, namely by adopting service composition patterns and automation to cater for resilience, scaling, and placement of service instances.

The specific nature of some MCN services that provide connectivity in or across geographic locations (e.g., RANaaS) makes them inherently different from purely cloud-native applications. The former can rely on the cloud provisioning model spawning new (virtual) resources only for compute and storage resources in the cloud and not for resources that are tightly bound to a specific hardware or geographic location. For the latter kind of resources traditional approaches to reliability (i.e., physical redundancy) and “scaling” (i.e., capacity planning) have to be in place and are still extremely important.

5 Technical Reference Implementation of MCN Architecture

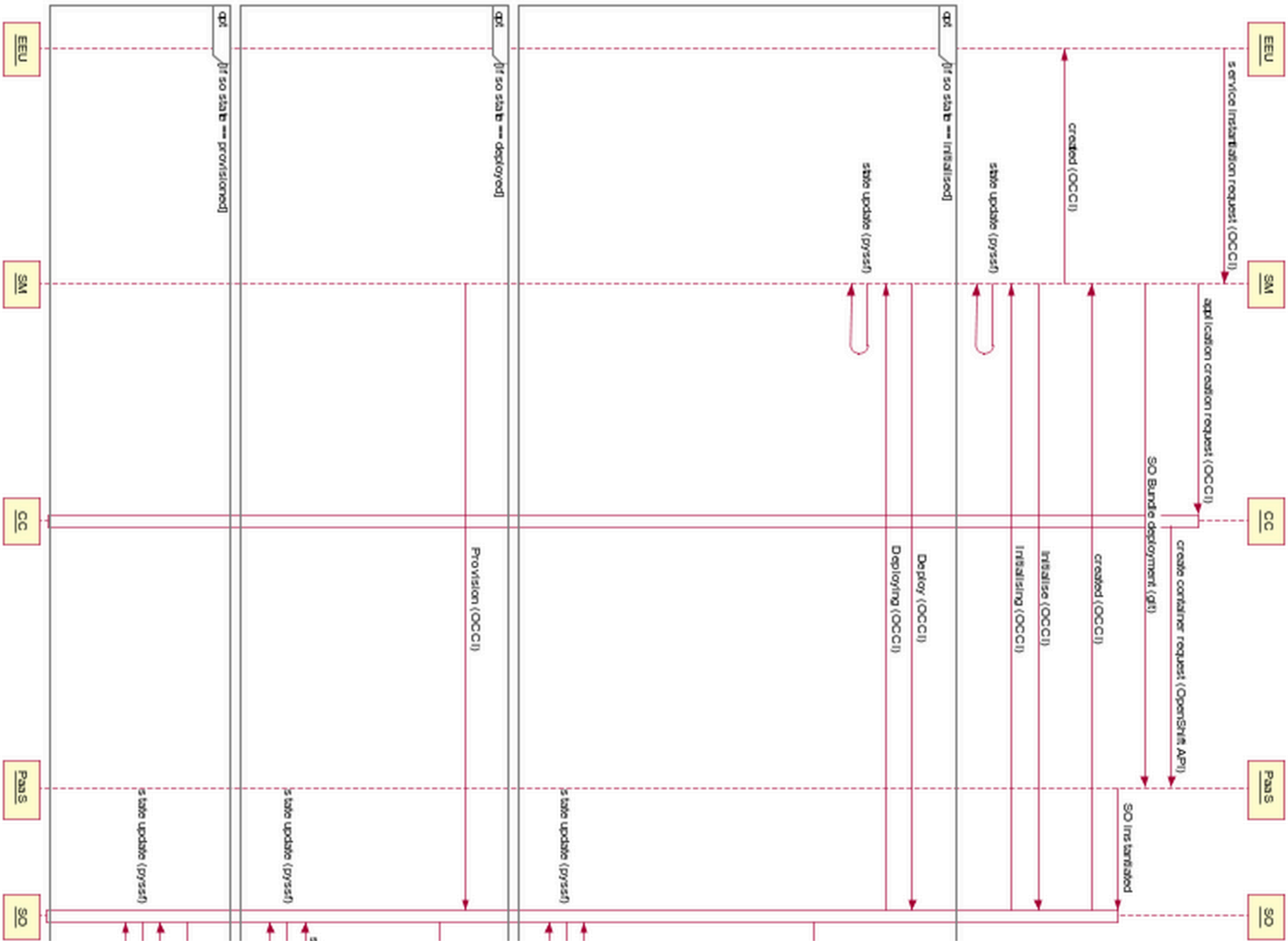
The reference implementation of the MCN has closely followed the prescribed architecture which was delivered as part of D2.2. In principle, the architectural elements are influenced by SOA principles, and constructs are in place to facilitate the service lifecycle management. The detailed representation of each architectural element is already shown earlier, each individually represented by a FMC diagram. In this section, we will analyze the technologies used to realize the full reference implementation of this architecture.



R
OCCI
Service Bundle
Service Manager (mcn_sm & pyssf)
R
OCCI
R
OCCI
SDK (Python)
Service Orchestrator (Python)
R
svc. specific
OCCI
R
OpenShift API
Cloud Controller
R
R
R
OCCI
R
svc. specific
Service Instance (many have N SICs)
PaaS (OpenShift)
IaaS (OpenStack, CloudSigma, Joyent SDC)

Figure 10 Technological Components of Reference Implementation

Figure 10 shows the static technology components' relations and use in the implementation of the reference architecture even though it shows a simple scenario with only one service depicted. In order to understand the interactions of these components please refer to the following sequence diagram (note: it is split in two for easier reading) which walks through the process of creating a service instance. Note that the interactions with the SO and its Resolver component are detailed in section 1.2.2.5.


EEU
SM
CC
PaaS
SO
service instantiation request (OCCI)
application creation request (OCCI)
create container request (OpenShift API)
SO Bundle deployment (git)
SO instantiated
created (OCCI)
created (OCCI)
Initialise (OCCI)
Initialising (OCCI)
state update (pyssf)
opt
[if so state == initialised]
Deploy (OCCI)
Deploying (OCCI)
state update (pyssf)
state update (pyssf)
opt
[if so state == deployed]
Provision (OCCI)
state update (pyssf)
opt
[if so state == provisioned]
state update (pyssf)
EEU
SM
CC
PaaS
SO

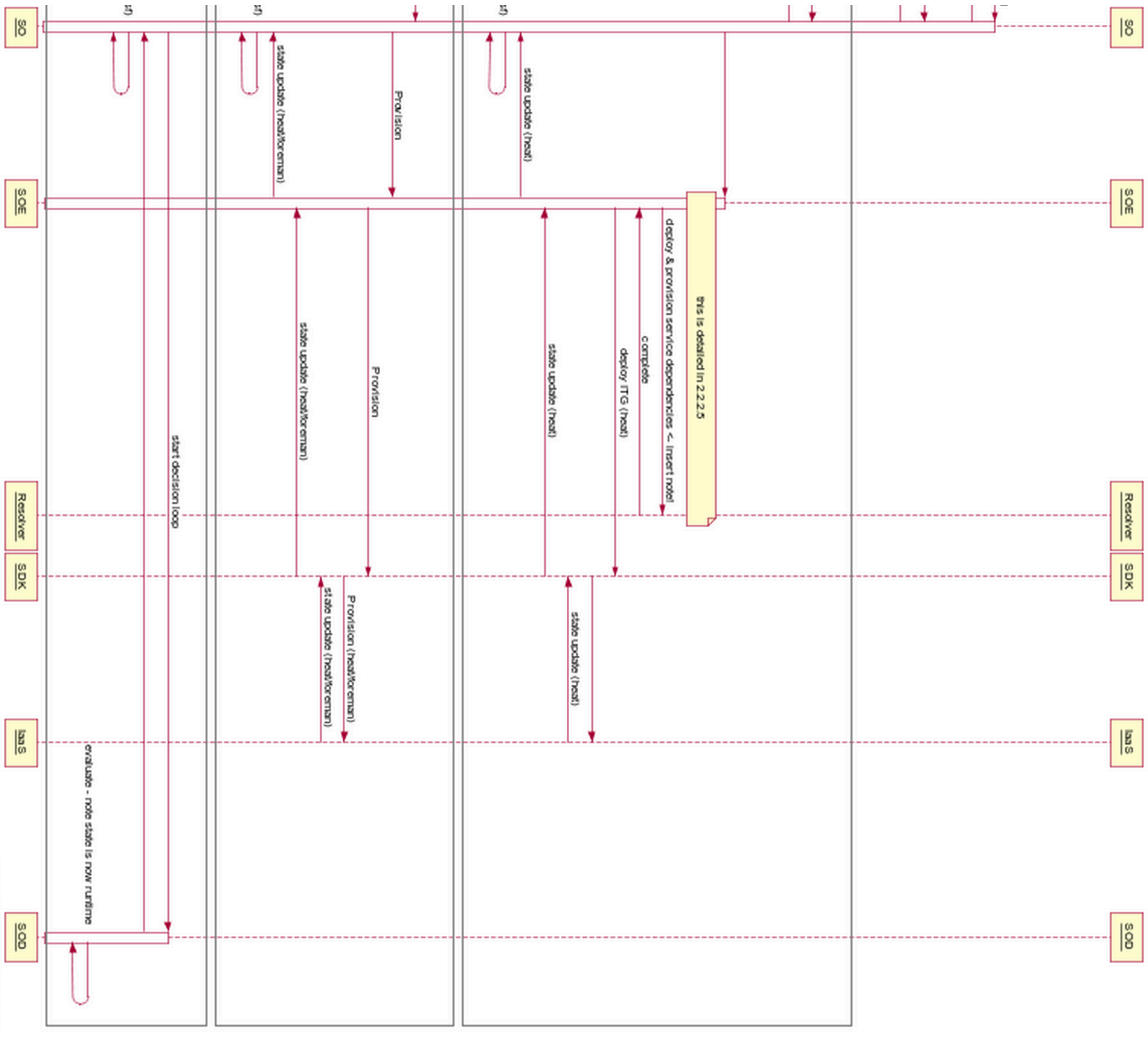

SO
SOE
Resolver
SDK
IaaS
SOO
state update (heat/foreman)
Provision
state update (heat)
this is detailed in 2.2.2.5
deploy & provision service dependencies <- insert note!
complete
deploy ITG (heat)
Provision (heat/foreman)
start decision loop
evaluate - note state is now runtime

Figure 11 Service Composition

5.1 Python

Most of the implementation was done using Python[10], which is generally very handy for a quick prototypical implementation. Python comes with a huge collection of libraries for virtually any task which helped a lot in the fast turnaround time for the architecture reference implementation.

Python was selected as the default language for implementing MCN components as it is an ideal language for fast prototyping of architectural concepts.

[10] https://www.python.org/

It should be noted that the reference implementation of the MCN architecture is not bound to python as OpenShift that executes the SOs has support for many languages including Java[11]. This would make is easy and possible to deploy a Java SO and there would be minimal changes to the reference implementation.

5.2 PySSF

The Python Service Sharing Facility library[12] is an OCCI compatible implementation that is available on pypi[13]. It provides an OCCI compatible frontend and provides a Python Web Server gateway Interface, WSGI[14], application can can run on most web frameworks. PySSF formed the core of the OCCI interfaces in MCN reference implementation, and helped us implement a uniform interface that brought interoperability among all the services developed within MCN.

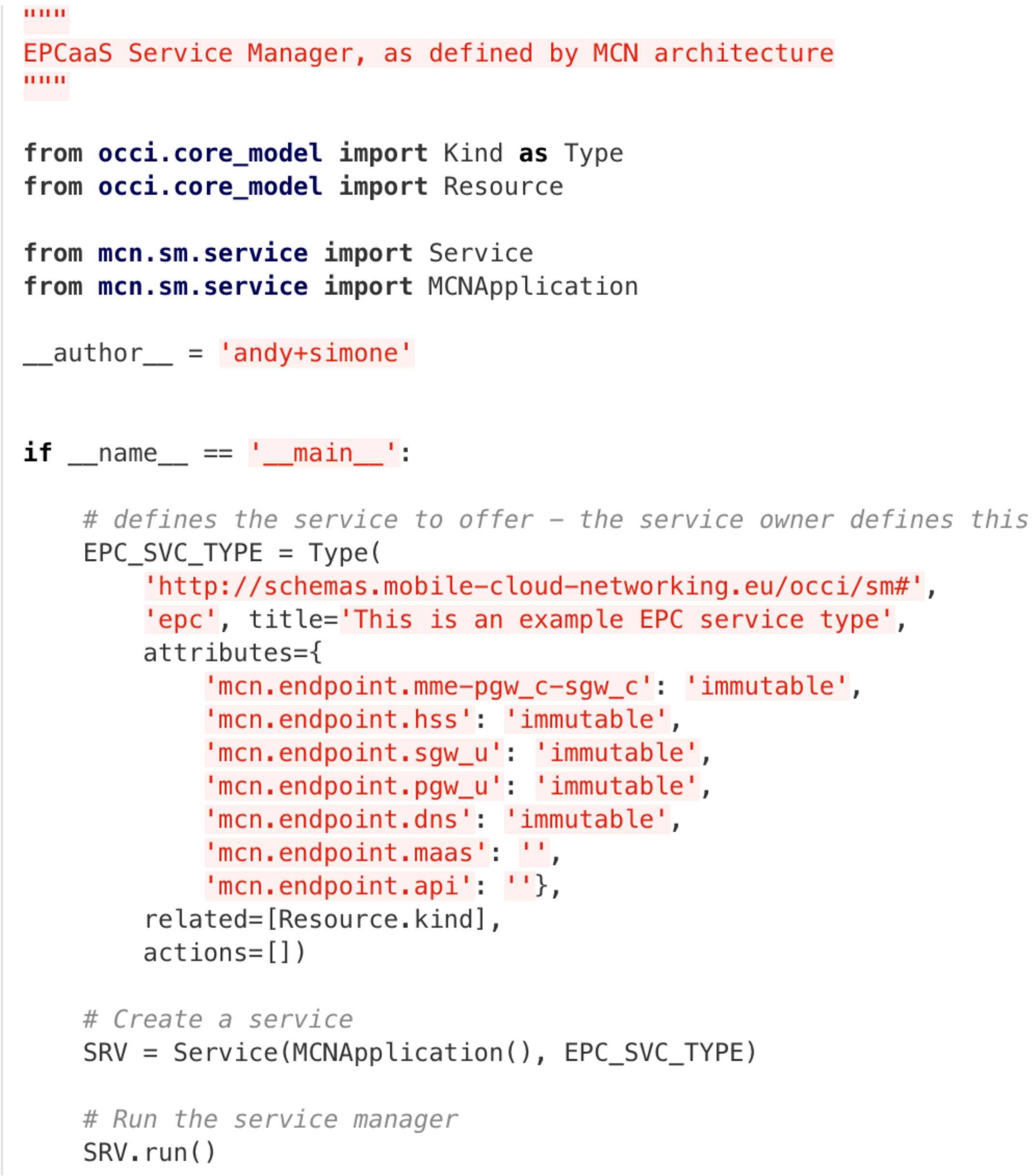
Each service is defined by extending the OCCI core model as implemented by PySSF, which provides a model that is easy to extend and provides interoperability out of the box. A typical service definition is shown below, in this case the EPC service.

[11] https://www.java.com/

[12] http://pyssf.sourceforge.net/

[13] https://pypi.python.org/pypi

[14] https://www.python.org/dev/peps/pep-3333/



```
"""
EPCaaS Service Manager, as defined by MCN architecture
"""

from occi.core_model import Kind as Type
from occi.core_model import Resource

from mcn.sm.service import Service
from mcn.sm.service import MCNApplication

__author__ = 'andy+simone'


if __name__ == '__main__':

    # defines the service to offer - the service owner defines this
    EPC_SVC_TYPE = Type(
        'http://schemas.mobile-cloud-networking.eu/occi/sm#',
        'epc', title='This is an example EPC service type',
        attributes={
            'mcn.endpoint.mme-pgw_c-sgw_c': 'immutable',
            'mcn.endpoint.hss': 'immutable',
            'mcn.endpoint.sgw_u': 'immutable',
            'mcn.endpoint.pgw_u': 'immutable',
            'mcn.endpoint.dns': 'immutable',
            'mcn.endpoint.maas': '',
            'mcn.endpoint.api': ''},
        related=[Resource.kind],
        actions=[])

    # Create a service
    SRV = Service(MCNApplication(), EPC_SVC_TYPE)

    # Run the service manager
    SRV.run()
```

Figure 12 Service Definition

In the figure above, it can be seen that a service type is given an identity (http://schemas.mobile-cloud-networking.eu/occi/sm#epc) and then has a set of parameters defined. These parameters can be easily discovered from the query interface that is supported from the deployed service manager.

Although the amount required to declare and service an interoperable and MCN-compliant service is rather minimal, this has been further simplified by the introduction of the service manifest, a technical output that is described in D3.4.

5.2.1 **OCCI**

OCCI, as explained in Section 2.1, forms the core of the interfaces exposed by the Service Manager, CloudController and the Service Orchestrators. Using an open standard allowed MCN service interfaces to be developed in an uniform manner, which maximizes interoperability, and saved a lot of time in development as the interface realization using PySSF was quick and painless. OCCI is also the specification used by the northbound interface of the CloudController.

5.3 Cloud Controller

The Cloud Controller is realized true a set of modules: design, deployment, provisioning, runtime and disposal. Each of the modules is modeled as a service itself. We defined those as the Cloud Controller's internal services as part of D3.1.

A preselection was made on which technologies could be used and extended in earlier deliverables. D3.4 gives a final overview of the selections made. The selections made are obviously abstracted away using the Service Development Kit and through the OCCI compatible interfaces. Hence a vendor/technology lock-in is not given.

For now the CC's internal services for the implementation of the CloudController are listed here:

- Design - Provided through reusing the OpenStack Keystone service.
- Deployment - provided by the PaaS OpenShift[15] for Service Orchestrators and OpenStack Heat for virtual resources. OpenStack heat was extended through plugins to support the different platform in place (OpenStack, Joyent SDC and CloudSigma).
- Provisioning - Realized by different technologies to provide a diversity of choices to the service developers. Realized in the easiest form through cloud-init functionalities passed through OpenStack heat. Ad-hoc changes can be done by sending commands through SSH, while the most feature rich solutions is provided through Foreman.
- Runtime - Provided by OpenStack Monasca[16] but is pluggable and can also be realized by other means.
- Disposal - related to both deployment and provisioning so implicitly pluggable

The NBI of the PaaS based deployment module is exposed as an OCCI interface and this interface definition has been submitted and accepted by the OCCI workgroup as "Open Cloud Computing Interface - Platform" and will figure as part of the OCCI 1.2 set of specifications.

5.4 Service Development Kit

The Service Development Kit was introduced to support service developers and those who write code to manage services. It server two main purposes:

1. Service Development Kit (SDK) supports the basic functionalities for the lifecycle management of any service by allowing the SOs to interact with the various internal services of the Cloud Controller (CC). Again changes to the internal implementation of the CC will have no impact on the code written, as the SDK abstract the technologies used.
2. Provide access to the support service developed within the MCN project. Focus here is on abstracting the concrete technologies away again. This will allow for service developers to e.g. interact with a DNS service is a certain way, regardless how the DNS service is implemented.

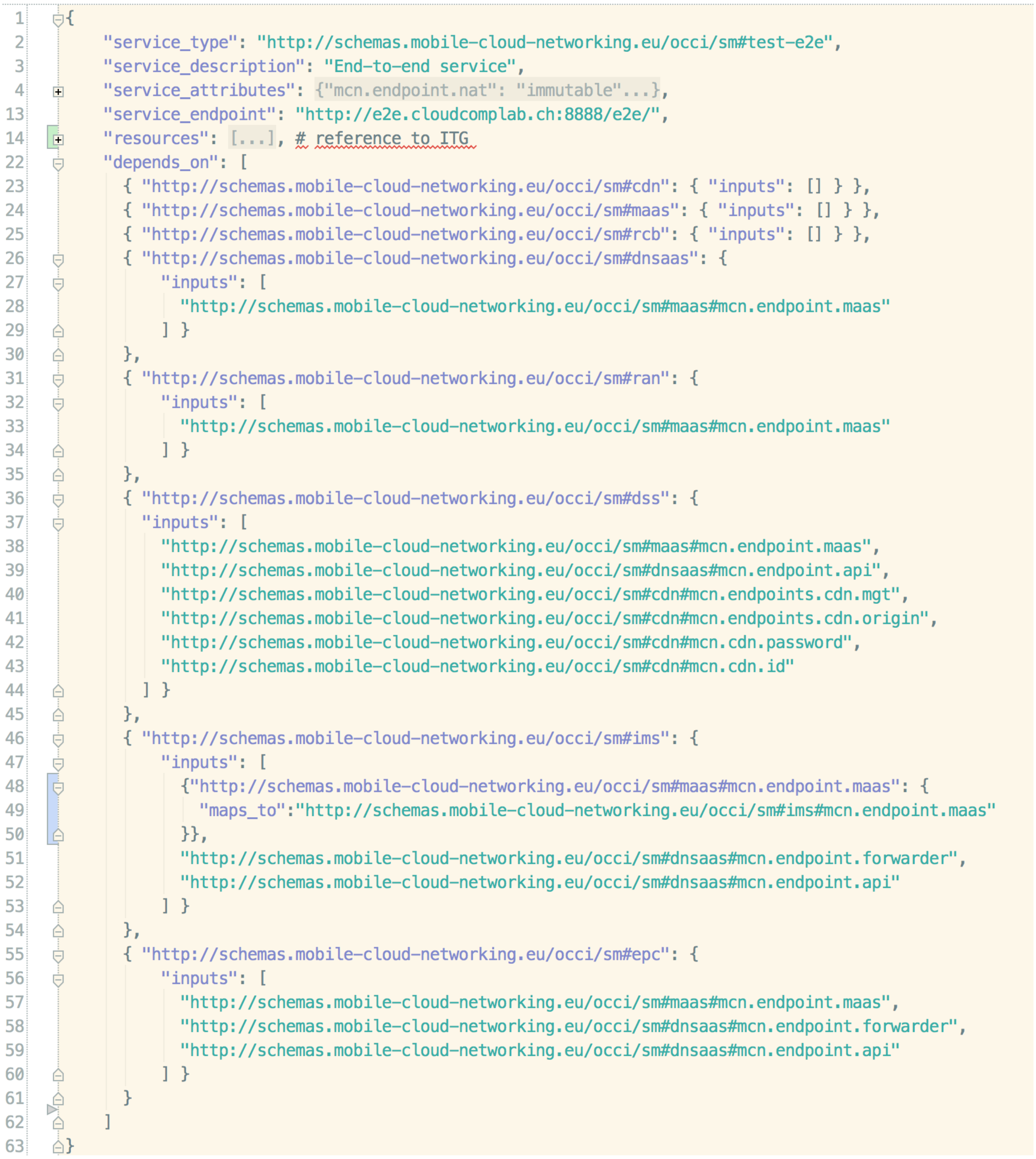
For more details, refer to the deliverable D3.4.

[15] http://www.openshift.org

[16] https://wiki.openstack.org/wiki/Monasca

5.5 Service Bundle

Every service, while implementing the Service Manager has to provide the Service Bundle which has all necessary details for the deployment. It contains the Service Manifest details, which comprises of Service Template Graph (STG) where dependencies on other external services are encoded, and Infrastructure Template Graph (ITG), which is realized as a Heat Template which encodes all the details for the deployment of the service itself. The service bundle also contains the service orchestrator logic, aka the application code. Below is an example of a Service Manifest.



```
{
    "service_type": "http://schemas.mobile-cloud-networking.eu/occi/sm#test-e2e",
    "service_description": "End-to-end service",
    "service_attributes": {"mcn.endpoint.nat": "immutable"...},
    "service_endpoint": "http://e2e.cloudcomplab.ch:8888/e2e/",
    "resources": [...], # reference to ITG
    "depends_on": [
      { "http://schemas.mobile-cloud-networking.eu/occi/sm#cdn": { "inputs": [] } },
      { "http://schemas.mobile-cloud-networking.eu/occi/sm#maas": { "inputs": [] } },
      { "http://schemas.mobile-cloud-networking.eu/occi/sm#rcb": { "inputs": [] } },
      { "http://schemas.mobile-cloud-networking.eu/occi/sm#dnsaas": {
          "inputs": [
            "http://schemas.mobile-cloud-networking.eu/occi/sm#maas#mcn.endpoint.maas"
          ] }
      },
      { "http://schemas.mobile-cloud-networking.eu/occi/sm#ran": {
          "inputs": [
            "http://schemas.mobile-cloud-networking.eu/occi/sm#maas#mcn.endpoint.maas"
          ] }
      },
      { "http://schemas.mobile-cloud-networking.eu/occi/sm#dss": {
        "inputs": [
          "http://schemas.mobile-cloud-networking.eu/occi/sm#maas#mcn.endpoint.maas",
          "http://schemas.mobile-cloud-networking.eu/occi/sm#dnsaas#mcn.endpoint.api",
          "http://schemas.mobile-cloud-networking.eu/occi/sm#cdn#mcn.endpoints.cdn.mgt",
          "http://schemas.mobile-cloud-networking.eu/occi/sm#cdn#mcn.endpoints.cdn.origin",
          "http://schemas.mobile-cloud-networking.eu/occi/sm#cdn#mcn.cdn.password",
          "http://schemas.mobile-cloud-networking.eu/occi/sm#cdn#mcn.cdn.id"
        ] }
      },
      { "http://schemas.mobile-cloud-networking.eu/occi/sm#ims": {
          "inputs": [
            {"http://schemas.mobile-cloud-networking.eu/occi/sm#maas#mcn.endpoint.maas": {
              "maps_to":"http://schemas.mobile-cloud-networking.eu/occi/sm#ims#mcn.endpoint.maas"
            }},
            "http://schemas.mobile-cloud-networking.eu/occi/sm#dnsaas#mcn.endpoint.forwarder",
            "http://schemas.mobile-cloud-networking.eu/occi/sm#dnsaas#mcn.endpoint.api"
          ] }
      },
      { "http://schemas.mobile-cloud-networking.eu/occi/sm#epc": {
          "inputs": [
            "http://schemas.mobile-cloud-networking.eu/occi/sm#maas#mcn.endpoint.maas",
            "http://schemas.mobile-cloud-networking.eu/occi/sm#dnsaas#mcn.endpoint.forwarder",
            "http://schemas.mobile-cloud-networking.eu/occi/sm#dnsaas#mcn.endpoint.api"
          ] }
      }
    ]
}
```

Figure 13 Service Manifest

The Service Manifest is a JSON document and of note is the section named "depends_on" (Line 22). In this section the service dependencies the offered service requires are listed. A service dependency is noted by the service type identifier and the set of the required attributes (identified by name) a service request needs. The resolution of these service types and the required parameters is carried out by the SO's Resolver component.

Upon receipt of a service creation request, the SM creates an application container in OpenShift, and when the container is ready, pushes the service bundle into the container and deploys it. Once the SO is ready, the various lifecycle phases of the service are activated depending on how the SO logic was written.

5.5.1 Realising the STG and ITG

From the perspective of the reference implementation, the STG is implemented as part of the generic service manager library and is part of the service manifest. The STG can be accessed through the Query Interface (QI) of the OCCI compatible interface of the SM (realized through pySSF). The SIG is represented through the linked OCCI resource entities within the RESTful OCCI compatible interface.

The ITG is implemented through the use of AWS Cloudstack or OpenStack Heat compatible templates. For the needs of multi-region deployments support of more than one ITG (one per zone) is supported. IIGs are managed within SO instance. Both the STG and ITG are deployed as part of the SO Bundle. Technical details of this is available in D3.4.

5.5.2 Deploying a SM and SO Bundle

Currently SMs and their related SO bundle has an easy means to be deployed. The SM includes the means to have it run and serve requests using the highly optimised Tornado[17] library. The SM and SO bundle can be deployed onto any platform that supports Python.

5.6 CloudSigma, OpenStack and SDC Infrastructures

OpenStack is the cloud management framework that allows the lifecycle management of virtual machines. It provides infrastructure cloud services, referred in MCN as atomic services, namely compute, storage and networking. The OpenStack project is one of the biggest community driven project, similar to the scale of the Linux project. Because of this fact, this platform was chosen for the MCN reference implementation compared to other IaaS cloud frameworks such as CloudStack, OpenNebula, etc. OpenStack features include some of the key support services out of the box. These are VPNaaS, LBaaS, DBaaS among others. And the Heat orchestration module allows deployment of MCN services as one logical unit instead of a collection of number of virtual machines. The set of python development libraries and availability of several automation tools and scripts helped tremendously in quick developmental cycles in the MCN project.

Once the initial prototype was tested with OpenStack cloud, the MCN project successfully integrated CloudSigma's cloud infrastructure and are almost finished with including support for Joyent's Smart

[17] http://www.tornadoweb.org/en/stable/

Data Center[18] (SDC). The glue for supporting these platforms was Heat. A Heat implementation for CloudSigma's cloud, and SDC has been developed by the consortium that would enable the MCN reference architecture to support multi-region, geographically distributed service deployments and management, and possibly support cloud federations in the near term.

[18] https://github.com/joyent/sdc/

6 Implementing the Architecture & Experiences

What follows in the next sections are the individual lessons learnt for each service implemented and used in MCN.

6.1 MCN Services

Below is a set of all the services that are either implemented or delivered out of MCN, using its conceptual architecture.

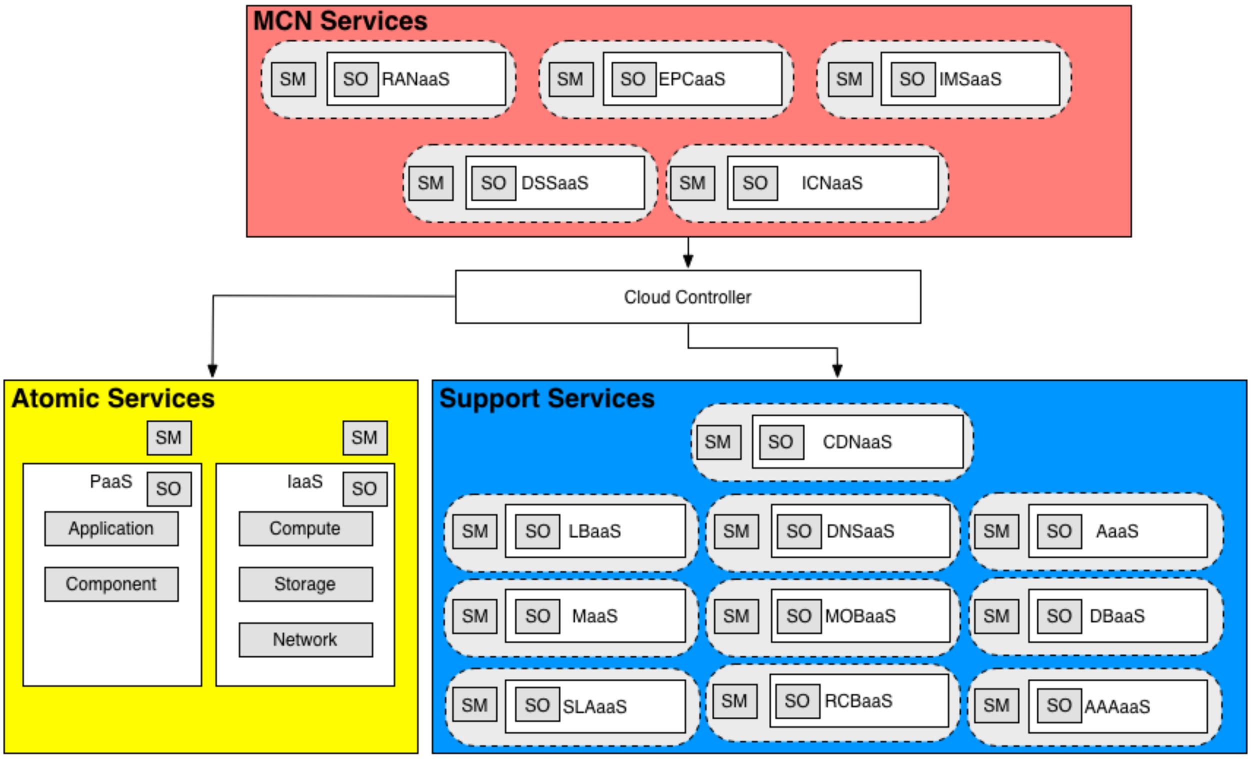


Figure 14 Services of MCN

6.1.1 RANaaS

RANaaS describes the service lifecycle of an on-demand, elastics, and pay as you go 3GPP RAN on the the top of cloud infrastructure. MCN architecture provided a good support for the development of the RANaaS SO/SM in view of achieving the required scalability and reliability (see Section 4.2). However, to support the scale in/out, RANaaS requires to adjust the maximum cell capacity and the resulted processing load to what is available. For this purpose, number of attached terminals, their context and stats have to be managed. In terms of reliability, the terminal and network contexts have to shared among different RANaaS instances.In the following, we summarize the lessons learnt during the development and experimentation of the RANaaS component:

- Processing time deadline:

  - FDD LTE HARQ requires a round trip time (RTT) of 8ms that imposes an upper-bound for the sum of BBU processing time and the fronthaul transport latency. Failing to meet such a deadline has a serious impact on the user performance.
  - Virtualized execution environment of BBU pool must provide the required runtime.
- Containers (LXC and Docker) vs Hypervisor (KVM)
  - Containers are more adequate for GPP RAN as they offer near-bare metal performance and provide direct access to the RF hardware. KVM performance is also good, but requires pass through mechanisms to access the RF hardware to reduce the delay of the hardware virtualization layer.
  - In case of containers, RAN requires low latency kernel in the host.
  - In case of full virtualization (KVM), hypervisor has to support real time/low latency task (different techniques requires for type 1 and 2), and also the guest OS requires low latency kernel.
  - Optionally, dynamic resource provisioning/sharing, load balancing to deal with the cell load variations (scale out and in).
- Hardware:
  - Management and sharing of hardware resources
  - Probe the existing RF for their capabilities (FDD or TDD, frequencies, transmit power, etc) and select the one that is required for the target RAN configuration.
  - support of RRH:
    - Frontahul (BBU to RRH link): a full-duplex 10Gbps link is required. There are certain ETH configuration to be done here.
    - Either EXMIMO 2 (PCI-x if), and/or NI/ETTUS (USB3 if)
- Flexible Configuration, build, run, and monitoring:
  - (Semi-)Automatic generation of the RAN configuration file through the UI or selection and editing of predefined configuration files. The same holds for the compilation (e.g. Rel 8 or Rel 10) and execution (e.g. enable disable hooks). Example of a config file can be found here[19].
  - Dynamic Monitoring of the status of RAN
- **IP address:** RAN requires to know the ip address of the available MMEs, S-GW, and P-GW.

Virtual Radio Resource Management: An important change in the RANaaS architecture was the introduction of the functionality of Virtual Radio Resource Management (VRRM)

[19] https://svn.eurecom.fr/openair4G/trunk/targets/PROJECTS/GENERIC-LTE-EPC/CONF/enb.band7.tm1.exmimo2.conf

as a part of SO, as described in deliverable D3.3. In a multi-tenant scenario, a single RANaaS SO instance can serve multiple EEU tenants using heterogeneous access infrastructures. EEU's requirements are characterised by an SLA. VRRM manages the resources to guarantee that each EEU's requirements are met. The shared SO contains information from all tenants without losing isolation between them. The SO is now capable of dealing with one or multiple EEUs. For it, a VRRM functionality was added to the SO, to compute the policies for management of the RAN, in order to guarantee EEU's SLAs. Also, an adaptation of the RAN scheduler was introduced to a) receive and apply VRRM's policies per EEU and to b) guarantee a feedback loop to VRRM, on the performance of the various EEUs. This enables to distinguish users of different EEUs, and enforce adequate scheduling policies through prioritization mechanisms.

All the functionalities of RANaaS are documented in D3.1, D3.2, D3.3 and D3.4. Final evaluation results are reported on D6.4.e

6.1.2 **EPCaaS**

Overall, the MCN architecture provided a good support for the development of the EPCaaS SO, proving that the initial design was overall sound and powerful. The "lessons learnt" during development of a MCN-compliant EPCaaS Service Orchestrator are summarized here:

- OCCI interfaces has been used for SM and SO interfaces, which simplified interaction with the supporting services and in particular facilitated service parameters passing from the SM to the SO.
- Particular attention was put into the synchronization of the different operations inside SO. For example, the start of the runtime phase (where the decision engine starts) must wait for the completion of the deploy and provisioning steps. Moreover, since the SM can send commands asynchronously, events generated by SM are checked continuously: e.g. the reception of a *dispose* command sent by the SM triggers the stop of the runtime phase.
- A basic service-specific decision engine has been developed, that triggers scaling-out/in based on pre-configured thresholds. However, the design of the EPCaaS SO was planned to be modular: the Policy Engine class, which encapsulated the decision making algorithm, enables coding of more elaborate decision algorithms and the use of external decision engines (e.g. like the one provided by the CC, e.g. Monasca), with no change to the rest of the code.
- Since the provisioning module in the CC was not ready and due to time constraints, some parameters were pre-provisioned in the service components. This has actually proved to be a very inflexible approach, since it supports only one pre-defined configuration of the service. Instead, operators want to be able to define their service parameters, specific for each deployment. The provisioning module in the CC will help, along with a flexible configuration mechanism on OpenEPC.
- According to the MCN Service Management Framework, the service components (e.g. the VMs) are created in the "deployment" phase and initially configured in the "provisioning" phase. But, if the service needs some parameters that rely on other Support or MCN Services (e.g. the IP address of a MaaS instance), created together with the main service, it must wait until these services are running and the parameters are actually available. If this

is the case, the main service cannot be *really* started in the runtime phase, until an *update* has been received by the SM, which provides the necessary parameters to the SO. The Update of a running service, i.e. a running OpenStack Heat stack, proved to be particularly challenging. Heat provides a mean (named “Software Configuration”[20]) to change the internal configuration of a running stack (i.e. the IIG, in MCN terminology) simply updating the stack template (i.e. the ITG). Unfortunately this convenient feature was unavailable: this forced us to implement an alternative mechanism, which forced the SO to wait for the first update by the SM (with all the correct parameters) to actually deploy the service.

- Regarding scalability, image size impacts the time needed to instantiate a new OpenEPC VM. This means that the time needed to complete a scale-out is longer. Lighter VM images should be used in the future, with only the needed modules and configuration files included
- Service Orchestrator is currently deployed on OpenShift PaaS. As this platform runs outside the OpenStack environment, VMs that need interacting with SO, for example for configuration purposes, must assign public IP addresses. If the number of VMs increases, this approach is obviously not feasible, since public IPs are a scarce resource. The OpenStack Heat feature mentioned above can represent a solution to this problem, since the SO only needs to send an updated template to OpenStack, to change configuration of running VM, without accessing them directly. In other cases this is unavoidable, especially when services are configured by means of RESTful APIs.

6.1.3 **IMSaaS**

Task 5.1 considered since the beginning of the project only the main signalling elements part of the 3GPP IMS architecture due to the limited amount of resources available for this subtask. In particular, they analysed and documented in the course of the first year of the project several architectural options in order to make the IMS infrastructure "cloud-ready”. Following cloud principles, three different deployment models have been proposed for the implementation of a cloud-based IMS platform and documented firstly in D5.1 [reference] and consequently in a paper (Magedanz, 2014):

- The first model is the Virtualized-IMS that is an implementation 1:1 of the IMS platform as standardized by 3GPP.
- The Split-IMS architecture, in which each network function has been implemented using a three-tier model typical for web services.
- The Merge-IMS is the last architectural model proposed, in which the network functions are grouped together and offered on a subscriber-based way.

In all of those models the scalability problem of the IMS infrastructure has been addressed. For instance, for the HSS has been proposed a separation between the front-end and the database, using cloud storage services, like the one provided by DBaaS (Section 6.2.8).

Some experiences have been gained while designing and developing all of those architectural options:

[20] http://docs.openstack.org/user-guide/enduser/hot-guide/hot_software_deployment.html

- Using heat in the deployment module in the Cloud Controller was a good choice. But, not having an abstraction on top of heat (for instance TOSCA and translated to Heat using the heat-translator[21]) was a limitation. The main motivation is that the SO developer was forced to produce heat templates as implementation of the ITG template for requesting resources to the cloud(s), requesting in this case a specific knowledge of what the different clouds are offering. An abstraction on top of heat would have simplified the way the SO requests resources to the lower layers.
- In order to retrieve the details of the deployed virtual resources it is required to declare the expected output parameters directly into the ITG template. It would have been better to receive from the Cloud Controller all the available information about the deployed resources, without having to specify them into a static template.
- The deployment of the SO on top of a different network than the one where the services are running reduces the way of communication between service instances and SO. In most of the cases, where the EMS interface is exposed by the service instances, it is required a floating IP in order to control them.
- Most of the functionality provided by the SOs implemented are similar for all type of services. Probably would have been better to increase the functionalities provided by the SDK creating out of it a ETSI NFV-compliant NFVO.

Additional features that could be useful for further developments:

- The Cloud Controller should integrate mechanisms from T3.1 for network function placement, exposing them to the upper layer (the SO) as high level requirements. For instance the SO might request network requirements in terms of high throughput and low bandwidth among two service instances. It is then the task of the cloud controller to select the datacenters where to deploy such virtual resources in order to meet those requirements.

All the functionalities of IMSaaS are documented in D5.1, D5.2 and D5.3. The evaluation results are reported on D5.4.

6.1.4 **ICNaaS**

Within the development of ICNaaS, several lessons were learned. These can be split into different types: solutions from the service to deal with architecture limitations and improvements to the development process brought by the underlying platform. Below we highlight a number of lessons of both types:

- There is no pre-defined approach to deal with communication between internal service components, given the implementation specific nature of each service. In ICNaaS, most operations are performed using RESTful web services, SSH and, naturally, the CCNx protocol for content delivery. At the same time, there was no standardized approach to perform communication between services, and service owners had to agree on a service to service basis.

[21] https://github.com/openstack/heat-translator

- As with OpenShift the Service Orchestrator is instantiated at a network, which is not the same as the one used by the service instances, all the service internal components need to have a public IP to be managed (for provisioning and reconfiguration operations). With the current limitation of IPv4 address assignments and the slow adoption of IPv6, this was frequently an issue for larger scale tests of the service.
- The versions of HEAT and OpenStack should have been defined at a given phase of the project (sort of a code freeze) and maintained afterwards. As this was not the case, updates to ITG and adaption of the use of certain version-specific features (e.g. network and floating IP assignment) had to be performed somehow frequently for ICNaaS orchestration components (every couple months). Such things taught developers a fair amount of lessons but also created some overhead and a few minor work peaks during critical periods, temporarily removing resources from the core development of the service (service internal components).
- With the MCN platform there was no common approach for service scalability decisions given the implementation-specific nature of services. The process of dynamically updating a service or part of it should be investigated. For ICNaaS, it meant that both a decision engine and a template generator had to be developed in order to handle service-scaling operations, however the new features of the CC runtime module can be considered for integration.
- As sometimes the documentation about the Service Development Kit and other resources to be used in the development of the service was not complete, due to ongoing development, for ICNaaS there was an extra effort of trial-error debugging and external code analysis in order to be able to proceed with the development of certain features.
- When considering the end-to-end scenario, it is positive that composed services can be easily created by a base service (e.g. DSSaaS), with the addition of the Resolver component.
- OCCI standard was a very good addition in terms of development simplification, transparency and abstraction. In fact, it should have been also implemented from the early stages in the SO without other temporary solutions.

All the functionalities of ICNaaS are documented in D5.1, D5.2 and D5.3. The evaluation results are reported on D5.4.

6.1.5 **DSSaaS**

A summary of experiences gained while development of a DSSaaS solution is provided here:

- According to the current MCN architecture, Service Orchestrator is deployed on through the CC. As this platform runs outside the IaaS service it is mandatory to assign a public IP address to each service instance component to reach them from SO. Alternatively, a proxy server could be configured to redirect requests to desired SIC.
- More development effort has been done to handle provisioning tasks as using init-cloud scripts is limited according to the architecture and the built-in provisioning module of CC was not ready. For DSSaaS an internal agent has been implemented for SICs to handle these tasks.

- The approach for handling internal communication between Service VMs was not defined as this is service specific. In case of DSS a RESTful interface has been implemented and used. Another possible option was using a messaging service like ZeroMQ. Inclusion of an ESB was discarded due to the complexity.
- OCCI interface has been used for SM and SO interfaces. Keeping the OCCI standard in all these communications and also simplicity of interacting with this interface reduced the complexity of development and testing of DSS SM and DSS SO interfaces.
- Also availability of the MCN SDK in combination with Heat made deployment and updating the service instances simple to manage.
- Automatic generation of and dynamic modifications on DSS service template management has been done by designing a specific module for DSS SO as a generic approach was not defined. Some other options has been evaluated like Troposphere library that was not finally used as it was specific to Amazon CloudFormation but might be useful for dynamic template generation.
- Decision engine for handling scales and dynamic adaptations of the service has been managed by designing and implementing a specific module for DSS service as this engine was not provided by the CC itself.
- Internal decision engine for handling scaling has been developed considering that it should be able to work as a standalone module or use an external decision engine provided by cloud controller like Monasca, which only provides part of the solution (it is still under consideration however).
- OpenStack Networking (Neutron) reduces the complexity of network configurations. The ability of defining security groups for OpenStack tenants made it much easier to manage network security and firewalls. Neutron Load Balancer as a Service simplifies the horizontal scaling procedures allowing you to attach new components to an already existing VIP. Thanks to the dynamic attachments and detachment of VMs to the Neutron LBaaS, it is possible for an OTT application to grant reliability to non-stateful components.
- Scaling out DB instances can be done by creating BD clusters but as Trove integration with Heat is currently poor and does not support clusters, a lot of effort is needed to manage (deploy/provision) DB clusters manually.
- Current scale up feature implies a down time for the scaled component so it is almost impossible to keep the service reliable just by scaling up a component.
- Scaling in implies some issues like state management of active sessions on the components that are going to be disposed. For an OTT application, keeping a live copy of the session information in an in-memory DB node across all the components can provide session persistence despite of scale-in processes.

Apart from the pros, cons and limitations described above, MCN architecture is flexible enough to address all the needs of a cloudified DSS service. Main DSSaaS achievements are described in the following bullets:

- The MCN platform allows two possibilities to demonstrate E2E.

  - First is DSS as a stand-alone service (managing composition internally). MCN SDK made it possible to request for another service needed by DSS, simply by requesting a service object. This service object can be used for all further interaction between the two services which to a high degree reduces the complexity of integration.
  - Second approach is to deploy all the required services by an external E2E SM then feed the DSS instance with the necessary EPs. Later DSS service can interact with any provided service using these EPs. Thanks to OCCI interface used for SM and SO, all these EPs can be pushed to DSSaaS by a simple OCCI call.
- Availability of the support services significantly reduces the amount of code of DSS service. Using simple interfaces we can benefit from:
  - MaaS to have all the service instances monitored.
  - AAAaaS simplified DSS authentication management.
  - DNSaaS for assigning the required domain names.
  - RCBaaS retrieves and processes billing information directly from MaaS.
  - DBaaS provides shared storage.
  - LBaaS simplifies scalability and provides fault tolerance.
  - SLAaaS provides service level agreement.
- Availability of the content delivery services CDNaaS and ICNaaS removes the necessity of deploying Content Cache Repositories (CCR) as a part of DSSaaS instance and at the same time increases the reliability and provides fault tolerance and smart content location for content delivery.

All the functionalities of DSSaaS are documented in D5.1, D5.2 and D5.3. The evaluation results are reported on D5.4.

6.2 Support Services

6.2.1 MaaS

Monitoring-as-a-Service (MaaS) addresses the design, implementation and test of monitoring mechanisms, from the low-level resources to the high-level services, across the four different domains: radio access network, mobile core network, cloud data center and applications. MaaS is considered as a full-stack monitoring system equipped with the capabilities to provide monitor and metering functionalities in a large scope of telecommunication systems. While developing MaaS within MCN, several lessons have been learned and are summarized in the following.

- The continuous process of updating the requirements from stakeholders on MaaS was the success for building a MCN service highly suitable for the project demands. A questionnaire has been initially used to derive required metrics from stakeholders, after which bilateral direct discussions followed and specified the requirements in more detail. This process has been kept during the implementation phase, in order to include changes in requirements as early as possible.

- The supporting service MaaS has been required from many other MCN services. Therefore a stable and specified reference points was required right from the beginning. First a static version with stable reference points has been released, later on an extended version has been released which supports the MCN life cycle.

- It turned out to be very important to have an early and well-documented prototype for handing over to the consortium for early testing and integration. MaaS has been the first MCN service that has been integrated with other services (IMSaaS and EPCaaS) within the project. A detailed documentation has been made available to the consortium for the use of MaaS.

- It has been recognized due the development that the use of MaaS would have required MCN service owner to perform similar developments for configuring and using MaaS. Therefore in order to avoid duplication of work, a decision has been made, to place such logic right into the SDK that is used frequently by other MCN services. Those functions are e.g. get_maas() and get_item(). As a result, a speed up in MCN service realization time, simplified deployment and usage has been achieved.

- A simplified process has been introduced for service owners to create and configure generic or individual service metrics. In addition, a subscription for requested metrics has been realized as well as the definition of threshold definitions for triggering an appropriate action (e.g. scaling in/out).

- Service stability of MaaS has been achieved by making use of solid and established open source projects such as Zabbix.

- Scalability of this supporting service has not been included in the DoW, but has been ensured partly by identifying potential performance bottlenecks and according resolutions. One of those potential solutions is the use of DBaaS within MaaS. More details on scalability and reliability can be found in the D34 section on MaaS.

- Challenges with MCN testbed infrastructure have been blocking the fluent development slightly. The ability to test MaaS on a single testbed made the availability of the testbed crucial. Due to version updates of OpenStack, developments of E2E SM and E2E SO, OpenShift updates, testbed capacity limitations, testbed availability in the end delayed the development, integration and test phase slightly.

- Several different monitoring systems exist which are directly or indirectly usable for OpenStack deployments. During the initial phase of the project, the Zabbix toolkit has been selected for MCN, with the motivation given in D3.1 and D3.2. During the runtime of the MCN project, new OpenStack monitoring systems appear. Those do not substitute MaaS or Zabbix in terms of functionality and stability, but this might change in the future in upcoming releases. In order to keep the agility of MaaS also for other/new monitoring systems, a design decision has been made for the support of heterogeneous monitoring systems with MaaS. Therefore a level of transparency has been introduced in the concept and its implementation of MaaS by abstracting the MaaS features from the open source toolkit Zabbix. The simplified implementation of new monitoring systems is therefore supported through implementing the main interfaces of MaaS. Tests of MaaS on the CloudSigma testbed showed that MaaS is operating with multiple cloud platforms including OpenStack, but MaaS is not limited to OpenStack.

All the functionalities of MaaS are documented in the previous deliverables D3.1, D3.2 and D3.3.

6.2.2 DNSaaS

The DNS as a Service (DNSaaS) allows to cloudify DNS and is employed in MCN as a support service from the early beginnings. The implementation experiences can be summarized as follows:

- The decision to include support services in the MCN CC Service Development Kit (SDK) should have been performed in early phases of the project. For services relying on DNS, it would have facilitated the integration with DNSaaS. The *get_dnsaas()* method allowed an easier integration with other services, such as EPCaaS, IMSaaS, DSSaaS, avoiding, as well, the use of repeated code.
- The versions of the deployment module and associated software, such as OpenStack and Heat lead to testing different methods for provisioning the DNSaaS, for instance to configure the IP of MaaS in the DNSaaS instance to enable resource and performance monitoring. This fact also required changes in DNSaaS to allow configuration of instances after their deployment. Different infrastructure template graphs were configured and tested (e.g. heat templates).
- The use of MaaS to support scalability issues was a good decision, considering the MCN architecture and the different services. With MaaS a constant performance monitoring is possible, which enable scaling decisions for DNSaaS not tied to a specific cloud infrastructure.
- The employment of OCCI facilitated the exchange of information between the service manager and the service orchestrator of DNSaaS, for instance parameters and the status of the service (i.e., if deployed or updated).
- The management of services composition was one of the latest features that was introduced. This fact also did not facilitate the integration of services with dependencies, however will in the future. For instance, this will facilitate the integration between DNSaaS and MaaS.
- The evaluation performed regarding scalability of DNS demonstrated that the chosen architecture for DNSaaS was reliable in terms of failures in the backend servers (PowerDNS Servers) and adapted to the load requested by DNS clients.

All the functionalities of DNSaaS are documented in D3.1, D3.2 and D3.3. The evaluation results are reported on D3.4.

6.2.3 CDNaaS

CDN-as-a-Service (CDNaaS) aims to improve the performance of, efficiency and ease of retrieving content in mobile networks. Its development has been stopped and it is considered feature-complete and ready to use by other services. The implementation experiences are as follows:

- CDNaaS is an exception compared to most of the services within MCN as it does not deploy any virtual resources: It has no infrastructure graph as part of its SO. Early in development, the choice has been made to avoid reinventing storage components when CDNaaS could instead fully exploit well established tools such as OpenStack Swift[22]. As a result, the added value of CDNaaS instead lies on the *management* of the underlying infrastructure, creating and correctly configuring accounts and user preferences on-demand. Looking back, this was the best choice to create an optimised service in line with the requirements of MCN.

- It could have been possible to use an object storage infrastructure directly from the provider, from instance using a CC call to retrieve an account and credentials, but it was not a requirement of testbeds to provide object storage in general, and Swift in particular. In that regard, it was decided at the time of the design of the service to completely manage Swift from the service perspective, rather than relying on the underlying MCN architecture. As service orchestrators implementation is completely up to the service developer, this was in line with the MCN framework and allowed us to create and manage its own instance of Swift distinct from the MCN testbeds.
- CDNaaS was initially envisioned as a full service, but after the first year architecture design, it was decided to instead view it as a support service. This choice was made as CDNaaS in the framework of MCN is not made to be deployed as a single service. It is not one of the traditional telco services which MCN aims to move to the cloud, but is an essential part for services which aim to distribute content efficiently to their clients, as such it fits the definition of a support service.
- Consumers for the CDN service are typically the main services of MCN, though the main user is the DSS service which has to distribute a high volume of content to clients. In hindsight, the service could thus have been tailored more specifically for DSS, for instance by integrating more content management policies such as customizable time-to-live according to types of content.
- Scalability of the service is handled directly by the storage backend, that is Swift in our implementation. While it removes some control from the service orchestrator which can not use the runtime module to monitor virtual resources, it leaves scaling and replication policies to the storage infrastructure which in the case of Swift has been optimized over the years and can be deployed in an optimized manner.
- While exploiting Swift, CDNaaS still offers customization and replication possibilities, as the CDNaaS central server can manage multiple Swift instances, offering automatic replication of content within Points of Presences chosen by the user at creation time. This makes scenarios such as service instance shutdown possible by offering the user a way to put the content to multiple locations. By exploiting DNS techniques, a service orchestrator using both the DNS service as well as CDN can also seamlessly offer a unique address for each object uploaded to the CDN instance, irrelevant of their actual locations, which is useful to redirect an end user to the closest location hosting the requested content.
- As no resources are actually deployed using the SDK, and thus no runtime control can be defined in the service orchestrator, scalability and reliability depends upon the CDNaaS

[22] http://swift.openstack.org

service provider. As per the architecture described in D5.2, both the frontend servers (which are in essence web applications) as well as the backend Swift need to be highly reliable. The former can be achieved using typical frameworks for availability of web services, e.g. HAProxy[23] in combination with multiple instances of the backend database, while in the latter case we rely on the architecture of Swift, where multiple copies of each file are stored in different locations, to provide for a reliable object storage service.

- CDNaaS is well integrated within the MCN framework, a CDN service instance can be created through an OCCI call to its service manager directly, but the latest version is also integrated with the Cloud Controller SDK with a *get_cdn()* method, offering an easy way for a service to instantiate a new CDN service.

It has successfully been demonstrated as part of the Year 2 review as part of the DSS end-to-end demo. With its main user being satisfied with the current version of the service and all required features having been implemented, we decided to stop further developments to focus on services which required more work. All functionalities are described in D5.2 with an update in D5.3.

6.2.4 **SLAaaS**

Overall, the lessons learnt during the design and development of the SLAaaS can be summarized in the following points:

- The use of the OCCI SLAs specification for the realization of the provisioning and northbound APIs resulted effectively in a rich and powerful interface for the SLA Management System.
- The integration of existing frameworks and solutions for certain functionalities (e.g. Intellect[24] framework for the Rules Engine and SAA for the SLO assurance) was necessary for the efficient development of the overall system.
- The Collector component had to select during runtime the appropriate monitoring interface for each SLO and metric defined in an agreement, based on the infrastructure a device is hosted. Such functionality is very important for the interoperability and extensibility of the implemented system.
- The SLAaaS is a wsgi python application based on the PySSF[25] framework with a Mongodb in the backend to support the persistence of the resource, thus the scalability and reliability of the system can be managed by assigning more resources to the SLAaaS host VM. More details about the performance and evaluation of the framework can be found on the D5.4.

The architecture of the SLAaaS has been slightly changed from the one presented in D2.2. Most of the changes are described in D5.3 and concern the removal of the agent mechanism and the feedback component, and the clarification of the SLA assurance and the measurements collection process. As explained in D5.3, the SLA agents' capabilities will be performed by the monitoring agents of the

[23] http://www.haproxy.org/

[24] https://pypi.python.org/pypi/Intellect

[25] https://pypi.python.org/pypi/pyssf

MaaS, thus there is no need of implementing an additional information collection module. What is more, the Feedback component has been eliminated from the architecture while the functionalities provided by it are incorporated within the SLA Rules Engine. Specifically, the feedback actions towards the RCBaaS and the Service Orchestrator are being given through the policy enforcement process within the SLA Rules Engine for each individual agreement. Finally, the Cloud Controller is depicted in the architecture of the SLAaaS while it provides the Aggregator with the necessary information for the correlation of each device in an agreement (e.g. VM resources) with the respective provider's infrastructure.

6.2.5 **RCBaaS**

The RCB as a Service (RCBaaS) allows to cloudify Rating Charging and Billing. RCBaaS is employed in MCN as a support service that takes as input the service consumption metrics, processes them, calculates the price to be charged to the user, and generates the invoice for payment. The implementation experiences can be summarized as follows:

- The availability of the MCN SDK in combination with CC deployer and provisioning module makes it possible to deploy, provision and update the service instances in a simple way.
    - For RCBaaS - an innovative charging platform, Cyclops[26] for cloud+ (cloud and any services offered over any IaaS cloud) services, the uniformity of development strategy made possible because of the SDK ensured that the Cyclops team just had to create the correct Heat template and base VM images, and the MCN framework allowed Cyclops to be offered as a service with minimal effort. For the SM implementation, just the declaration of service specific attributes was all that was needed. The SO was realized by essentially re-using the sample SO implementation provided by the MCN WP3. The development effort for ensuring Cyclops-aaS was minimal and was as expected from the MCN framework. It is important to point out here that the Cyclops framework itself has no dependencies on any of the main or support MCN services, which probably lead to minimalistic effort on our development teams, and made ample time available for the development of the service's core features.
- Monitoring as a Service (MaaS) is used for collecting charging data coming from MCN services (like DSS i.e.) through MaaS charging agent. The integration of MCN services could be easily extended beyond DSS by adopting the MaaS charging agent.
- The CCC make available methods for provisioning the RCBaaS. For example, in case it is needed to configure the IP address of the MaaS in the RCBaaS instance in order to enable resource and performance monitoring.
- In order to retrieve the details of the deployed virtual resources it is required to declare the expected output parameters directly into the ITG.

[26] http://icclab.github.io/cyclops/

- The decision engine for scaling and dynamic adaptations of the service has been managed by designing and implementing a specific module for RCB service as this engine was not provided by the CC itself, however the runtime module of the CC can be considered for integration.
- A limitation arising from the use of third-party software (JBilling[27]) for generating invoices is that a set-up of the JBilling plug-in is due for the configuration of each new MCN services to be integrated in RCB.
- A further evolution of the service in terms of reliability could be provided by using the DBaaS that could replace the internal DB used to store charging data before sending to billing system.

All the functionalities of RCBaaS are documented in D5.1, D5.2 and D5.3.

6.2.6 AAAaaS

The AAA as a Service (AAAaaS) allows to cloudify Authentication and Authorization procedures in the MCN framework. The AAAaaS is classified as a support service in MCN and can be instantiated through its SM.

The service is based on the OpenAM[28] Identity Management (IdM) framework and it offers an authentication mechanism for seamless access to Web services in Single Sign-On, based on OpenAM Policy Agents.

Registration to the service is allowed only to users that are valid Telco subscribers, by verification of users' private identities (like IMSI or IMPI) derived from HSS. The user’s subscription profile is extended with attributes belonging to the Telco domain that allows the inter-working of digital identities in both the IMS and Internet domains thus making it possible to support more sophisticated business scenarios.

The OpenAM framework has been extended for adding user's profile attributes. The application framework has been designed in order to expose information related to the IMS/Telco domain (e.g. subscriber’s info from HSS or presence status) to 3rd parties via standard web APIs. The experiences derived from the design and the implementation of the AAAaaS service can be summarized as follows:

- Reliability is strongly related to the communication with the HSS. The service has been designed in order to work regardless the availability of the HSS during the authentication phase. On the other hand, in case the HSS is not reachable, it is not possible to check whether the user is a Telco subscriber, thus invalidating the registration phase.
- Scalability has not been taken into account for this service because it is strongly related to HSS behaviour that in principle could scale independently. In order to set up an efficient mechanism, HSS and AAAaaS should scale in a coordinated way that is not currently supported by the MCN framework. However, some evaluations will be carried out in order

[27] http://www.jbilling.com/community

[28] https://forgerock.org/openam/

to assess the performance of the AAAaaS service to cope with increasing workload (as foreseen in D5.4).

- Integration of MCN services could be easily extended beyond DSS by adopting suitable OpenAM Policy Agents for Single Sign-On. However, this implies the installation of the policy agent software and manual configuration for each federated service.

All the functionalities and details of AAAaaS are documented in D5.1, D5.2 and D5.3.

6.2.7 **MOBaaS**

Many lessons/experiences were collected during/after the development process of Mobility and bandwidth prediction as a service (MOBaaS). These can be further divided into three different types according to the development tasks: issues about the service itself, issues about the service deployment on the test-bed, and issues during the phase of integrating MOBaaS with other MCN services. Details of each lesson are listed below:

- The introduction of the OCCI mechanism simplifies the interface between SM and SO. This helps to transmit service specific parameters between SM and SO.
- After the implementation, a service has to be integrated with other MCN services. The use of OCCI and now soon in the Resolver implementation of the SO will enable such integration of other MCN services. However, for the integration of service instance components this still remains a service specific task and it would be helpful if a common means would be investigated.
- MOBaaS requires certain amount of historical user movement data to make prediction. This data is supposed to be ready at MOBaaS service instance before receiving any prediction request such that whenever a request is received, the prediction can be made out of the data. However, the big size of local-stored data will increase the size of the image, which will increase the time to instantiate a new MOBaaS VM when scaling operation is needed. Future plan is to remove the local data storage and to use DBaaS as the storing medium, which will make the scalability operation more smoothly.

6.2.8 **DBaaS**

Database as a Service required no development efforts on the behalf of MCN. It’s logical architecture was presented in D2.2 and has not required any update. DBaaS was provided by the OpenStack ecosystem. The implementation used was Trove[29]. There has been no further and additional development of Trove as it suited the basic needs of other services.

One thing that made Trove initially more difficult was the limited means to install it, however this has improved greatly over the past months, especially with the release of a puppet module[30] and

[29] https://wiki.openstack.org/wiki/Trove

[30] https://github.com/stackforge/puppet-trove

integration with packstack[31] one of the OpenStack deployment tools used on some of the testbed. Scaling out DB instances can be done by creating DB clusters in Trove[32].

Another alternative solution for the DBaaS implementation would have been to use the database services offered by OpenShift (this is already used to provide a runtime environment for SOs), however due to the limitation in OpenShift version 2 of not being able to publicly expose (IP address) these services this approach was not chosen. The DBaaS is used mainly within the DSSaaS and IMSaaS.

6.2.9 **LBaaS**

Just as with DBaaS, Load Balancing as a Service required no development efforts on the part of MCN. It's logical architecture was presented in D2.2 and has not required any update. LBaaS is provided by the OpenStack Neutron core project. The actual implementation used is the default one, HAProxy[33], which can load balance all IP-based communications. It should be noted that this particular load balancer is typically used for web and over-the-top type applications and has limited applicability within the services of RAN, EPC and IMS. The details of this are reported in D4.2, D4.3.

6.2.10 **Analytics Service**

The analytics service is a generic cloud-native service which allows for running complex machine learning/data analytics algorithms in the cloud. In doing so it allows specifically for deeper insights into system and performance behaviour based on the USE methodology[34], infrastructure instance graphs as well as monitoring data. A detailed description of the service can be found in D3.3.

It has deemed very helpful to have a service which enables the investigation of the service performance in correlation with the elastic topology of the infrastructure. Although still a topic which can be expanded upon the USE methodology has deemed very helpful here.

The fact that the analytics service is itself cloud-native allows for a scalable system. The Micro-Service architecture (realized with help of docker containers) allows for the creation of e.g. Complex Event Processing (CEP) based systems where continuous running analytics algorithms are realized. While this might require some compute power based on the algorithms run, we can unburden service orchestration instance. Instead of doing the calculation itself the SO instance can now be signalled by the analytics service. Having the loosely coupled systems also allows to high reusability of the analytics service in different scenarios.

6.3 Atomic Services

6.3.1 **IaaS**

Infrastructure as a Service (IaaS) in MCN remains to be provided by OpenStack and CloudSigma. As a result of work carried out in WP3, MCN can now also support virtual resources hosted on the

[31] https://github.com/stackforge/packstack/

[32] https://www.mirantis.com/blog/using-openstack-database-trove-replication-clustering-implementation-details/

[33] http://www.haproxy.org/

[34] http://www.brendangregg.com/usemethod.html

SmartDataCenter platform. Other than various means to deploy IaaS at different testbeds, nothing significant has changed. The key reason for this is primarily due to the abstraction offered by both the Heat Orchestration Template language and secondarily the abstraction offered by the Cloud Controller. This is a feature that is highly appreciated and has save much development effort. What modifications to this foundational service type is reported in D3.4 and details of how it's deployed upon testbeds is detailed in D6.4.

The interconnection between resources across data centers relies on secure mechanisms, such as VPNaaS that can be established dynamically when required as defined in the ITGs. On the other side, the intra DataCenter connectivity relies on OpenDaylight and on Neutron plugins that allow this one to act as OpenFlow Control Adaptor. The APIs and components allowing such connectivity are documented in D3.3 and D3.4.

6.3.2 PaaS

In order to execute SOs in an isolated and tenant-based environment a PaaS framework was chosen for the reference implementation. The technology selected was OpenShift which can provide the required features. This was a significant time saving choice as no work was required to develop the framework to support the execution of SOs. By taking this approach, the option to implement SOs in any language that the PaaS framework, coupled with OCCI RESTful interfaces, supported was also possible. Details of how it's deployed upon testbeds is detailed in D6.4.

7 Revised View of MCN and ETSI NFV

ETSI NFV Industry Specification Group (ISG) was set up in November 2012, coincidentally around the same time the MCN project was taking its first baby steps. Being a closed working group, the activities of most of the working groups were not in public domain until their first documents came out in the form of the white papers around October 2013. MCN project's overall architecture specification came out in the form of D2.2 in October 2013 too. Even though the two activities were happening in parallel, the architectural specifications from MCN as well as ETSI NVF ISGs were aligned with each other in most of the aspects except the nomenclatures used to refer to different architectural elements.

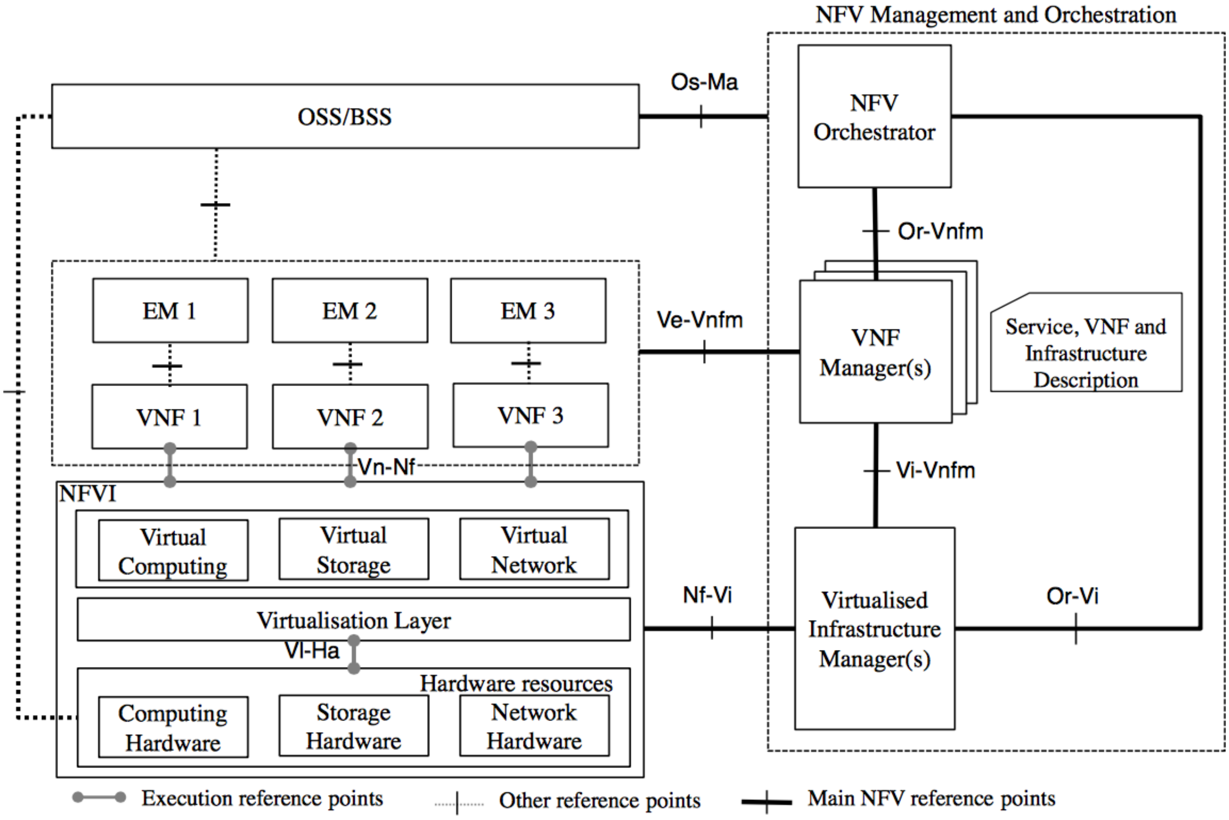


Figure 15 ETSI NFV Reference Architecture

Figure 15 shows the ETSI NFV reference architecture framework taken from the latest available reference document (GS NFV-002). We looked into the description of each architectural element in GS NFV-002 reference document, along with the various interface descriptions (actions supported by the interfaces, information flow through it, etc), and used the two sets of information to map ETSI architectural elements to the corresponding MCN components. This functional mapping is shown in Table 2.

Table 2 ETSI NFV Mapped to MCN

ETSI NFV architectural element	Functionality	Corresponding element in MCN architecture	Functionality	Exact Map (y/n)
NFVI	NFV Infrastructure: Allows for NFVs to be deployed, executed and managed.	Cloud data centers providing compute, storage and network.	Allows for MCN service to be deployed, executed and managed.	Y
VIM	Manages the interaction of a VNF with computing, storage and network elements	Cloud Controller + Cloud Managers such as OpenStack	Manages the interaction of a MCN service with computing, storage and network elements	Y
VNF Managers	Allows for VNF lifecycle management, instantiation, update, query, scaling, terminations	Service Orchestrators	Allows for main services lifecycle management, instantiation, update, query, scaling, terminations	Y
Service, VNF and Infrastructure Description	Data-set contains VNF forwarding graph, deployment templates, etc.	STG: Service Template Graph + Infrastructure Template Graph	Allows service components to be described and the deployment order, geographical placement hints, etc.	Y
OSS/BSS	OSS/BSS of an operator	AAAaaS + RCBaaS & various other MCN support services	Provides on demand OSS/BSS components along with monitoring, load balancing and other MCN services.	Y
NFV Orchestrator	In charge of orchestration and management of NFVI and realizing network services on NFVI, NFV-O sends configuration information to VNF-M.	Service Managers and related SO.	Allows for state and endpoint exchange between the MCN services and users, also passes the configuration hints to the service orchestrators for lifecycle management.	(N)
Element Management	EMs in ETSI can perform scaling and load balancing of various VNFCs	Each service in MCN implements its own scaling logic which manages each "element" i.e. a service instance component.	Allows scaling decisions of MCN service instance components based on monitoring data	(Y)

The MCN architectural elements largely remains unchanged from the previous iterations, and the ETSI VNF reference architecture also has not changed, thus apart from the lexical differences, the two architectural frameworks remain largely compliant with each other, with elements of the two mapped one-to-one in most cases as the table above demonstrates. The significant difference being in the NFV Orchestrator elements, which is the single functional entity in ETSI, whereas in MCN there are service managers, one for each type of service. This approach in MCN allows for better scalability of the overall orchestration experience, as well as removal of a single point of failure. A slight difference is in how the Element Management (EM) is represented in ETSI reference architecture as an external element with which the VNF-Manager interacts, in MCN architecture, the EM functionality is embodied within the Service Orchestrator manifested as sub-components SO-D and SO-E.

The ETSI NFV ISG Phase 1 completed in December 2014, with the industry specification group now in its phase 2, is focusing on evaluations of the proof of concepts, integration of SDN in the NVF architectural framework, security and interoperability aspects. The MCN consortium having adopted OCCI in all the interfaces between the various elements of the architecture, have already taken a big stride towards supporting interoperability, and the consortium with its demonstrators is ready to provide valuable feedback to ETSI in this aspect, if needed.

8 Conclusions

This deliverable presented the updated version of the MobileCloud Networking logical architecture that was created by all participants of Task 2.3.

Resulting from numerous architectural discussions, the architecture as presented in D2.2 is largely unchanged as little modification was required. Where modifications have been required these are noted within the document. The process around any updates to the logical architecture was discussed and detailed.

This deliverables also includes a view upon the reference implementation of the architecture and an updated evaluation of the logical architecture against the current ETSI NFV architecture. Through the implementation of the architecture, experiences and learnings of the individual services' are also detailed in the deliverable. These learning are expected to be further understood through the evaluation phase that is reported in D6.4.

More broadly, from our work in architecture, we can say that although the MCN architecture and services were designed before the microservices design pattern came to be, MCN architecture and services still follow the principles service-oriented architecture and cloud-native applications, namely by adopting service composition patterns and automation to cater for resilience, scaling, and placement of service instances.

However MCN architecture and usage of cloud viewed from a NFV perspective is different. Whereas NFV leverages the use of frameworks like OpenStack the focus is upon the virtualisation of what was once ran on a physical device, MCN goes beyond this and asks question about how best the implementation of that network function might be designed better with cloud native applications, microservices and SOA approaches.

We foresee that future implementations of Services use the Microservice paradigm and this indeed is the industry trend (for example netflix, soundcloud, halo are all early adopters of this). This can be very powerful to example to move service instance components within a mobile network from a data center towards the edge for performance and agility reasons.

We also foresee that it would be beneficial if the core services themselves to applythe Microservice paradigm. This might not always be possible based on existing and legacy systems and protocols but for new service instance components as well as service instance components being replaced this might be of highly value. This is a step-by-step approach which is started by decomposing the service instance components (applying cloud-native design patterns) into smaller parts within the MCN project (e.g the work of WP4; D4.1-4). And we foresee this trend going on and eventually leading to a Microservice based edge-to-edge cloud based mobile core network.

In the summary of D2.2 it was said that "it is foreseen that other services can fit into this architecture" and we can safely state that this prognosis was correct. Of the work noted to continue in D2.2, was to develop and understand the data schemas required for the services such that interoperability between all services is ensured. This was achieved by the adoption of a current Cloud Computing standard, OCCI and the experiences of implementing this were positive. Looking towards the future the relationship of MCN to a burgeoning area of work, namely cloud native application design, was detailed and also key recommendations on developing MCN services from the perspective of scalability, reliability and placement was reported.

References

Note: what is in **bold** is the reference key.

- Bill **Wilder**. Cloud Architecture Patterns. O'Reilly, 2012.
- Christoph **Fehling**, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. Cloud Computing Patterns. Springer, 2014.
- Alex **Homer**, John Sharp, Larry Brader, Narumoto Masashi, and Trent Swanson. Cloud Design Patterns - Prescriptive Architecture Guidance for Cloud Applications. Microsoft, 2014.
- Nicolas **Bonvin**, Thanasis G. Papaioannou, and Karl Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10, page 205, New York, New York, USA, June 2010. ACM Press.
- Adrian **Cockcroft**. AWS Re:Invent - High Availability Architecture at Netflix. - http://www.slideshare.net/adrianco/high-availability-architecture-at-netflix , Nov 2012. Retrieved 2015.03.16.
- Phillipa **Gill**, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. SIGCOMM Comput. Commun. Rev., 41(4):350–361, August 2011.
- Antonio **Filieri**, Carlo Ghezzi, Alberto Leva, and R Ole. Reliability-driven dynamic binding via feedback control. In SEAMS, volume 2, 2012.
- Galia **Shabtai**, Danny Raz, and Yuval Shavitt. Stochastic service placement. arXiv preprint arXiv:1503.02413, 2015.
- Michael T **Nygard**. Release It!: Design and Deploy Production-Ready Software. Pragmatic Bookshelf, 2007.
- **Grance**, P. M. The NIST Definition of Cloud Computing. NIST special publication, 2011.
- A. Edmonds, T. Metsch, A. Papaspyrou, and A. **Richardson**, "Toward an Open Cloud Standard," IEEE Internet Computing, vol. 16, no. 4, Jul. 2012.
- Carella, G.; Corici, M.; Crosta, P.; Comi, P.; Bohnert, T.M.; Corici, A.A.; Vingarzan, D.; **Magedanz**, T., "Cloudified IP Multimedia Subsystem (IMS) for Network Function Virtualization (NFV)-based architectures," Computers and Communication (ISCC), 2014 IEEE Symposium on , vol.Workshops, no., pp.1,6, 23-26 June 2014