Chapter 8
Deployment of Cloud Supporting Services

Gabriel Iuhasz, Silviu Panica, Ciprian Crăciun and Dana Petcu

8.1 Introduction

The main emphasis in this chapter is on the various supporting services needed to run an application. In the MODAClouds context, all services and resources involved in running and managing an application on a given Cloud provider comprise the runtime environment.

We give an overview of the Execution Platform (Energizer 4Clouds) and its main components and services that have a direct role in deploying the supporting services. In particular we will detail the mOS operating system and its main subsystems as well as the supporting services. We briefly talk about how all services are packaged and deployed after which we give an overview of and rational behind their design and implementation. These supporting services are: **Object Store**, **Artifact Repository**, **Load-Balancer Controller** and finally the **Batch Engine**. A brief overview of how the supporting services are used in the MODAClouds project will be covered at the end of this chapter. We also cover the runtime platform integration and interdependencies of the supporting services and various other platforms that comprise the runtime platform.

G. Iuhasz (✉) · S. Panica · C. Crăciun · D. Petcu
Institute e-Austria Timisoara, West University of Timişoara,
B-dul Vasile Pârvan 4, 300223 Timişoara, Romania
e-mail: iuhasz.gabriel@info.uvt.ro

S. Panica
e-mail: silviu@info.uvt.ro

C. Crăciun
e-mail: ccraciun@info.uvt.ro

D. Petcu
e-mail: petcu@info.uvt.ro

© The Author(s) 2017

E. Di Nitto et al. (eds.), *Model-Driven Development and Operation of Multi-Cloud Applications*, PoliMI SpringerBriefs,
DOI 10.1007/978-3-319-46031-4_8

8.2 MODAClouds Execution Platform

In this section we focus on the functionalities of the execution platform, and more precisely on the supporting services which enable the deployment and execution of various other services that are part of the runtime platform. In particular the runtime platform is responsible for monitoring and self-adaptation.

Figure 8.1 offers a general overview of the overall dependencies between the execution platform, the monitoring (Tower 4Clouds), adaptation (SpaceOps 4Clouds)

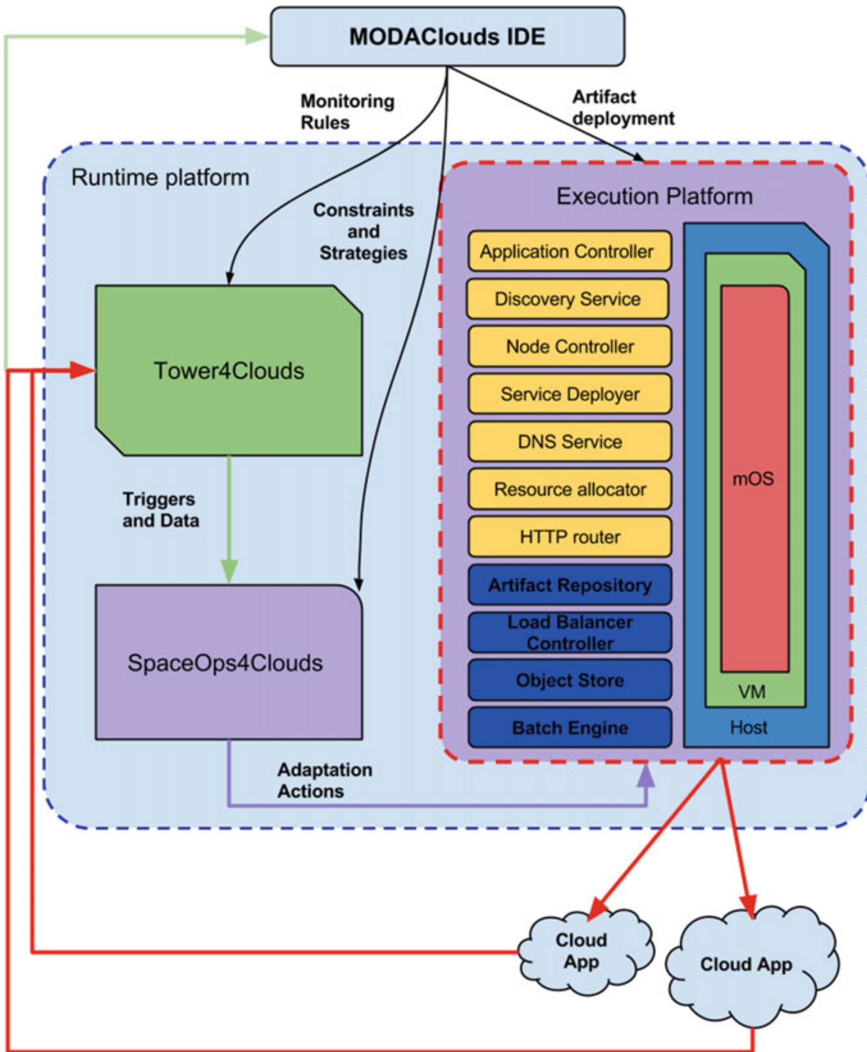

Fig. 8.1 Energizer 4Clouds—Execution Platform

and the MODACloud IDE. The execution platform has three main sub-systems; infrastructure, coordination, platform. The infrastructures sub-system handles low-level management of Cloud resources, coordination sub-system enables services to find one another and exchange messages and finally the platform sub-system handles the MODAClouds-specific tasks. The supporting services which are the main focus of this chapter can be found at the bottom of the above figure.

Discovery is an important functionality, required by all services from the execution platform. Each component consumes or provides various services, which are accessed in almost all cases over established networks protocols (HTTP, AMQP, raw TCP etc.). Thus, the developer is provided with API's that abstracts and expose these service endpoints and the way to resolve them.

8.2.1 mOS

The mOS operating system is based on existing open-source operating systems and it is used to host the MODAClouds Platform. Currently there are two versions mOS v0.x (based on Slitaz) and v1.X (based on OpenSUSE) [4]. It is designed to run on any compatible Cloud infrastructure. The pre-compiled kernels are available to support major Cloud providers such as Amazon EC2, Google Compute Engine and Flexiscale to name but a few.

There are several important services that run inside mOS which are paramount to its functioning. The mOS bootstrap service is tasked with customizing the execution platform by starting required services at boot time. These services are in charge of various actions that create the run-time environment. Other notable services are the so-called ZeroConf services which are special services hosted by the Cloud providers to enable the interaction between active VMs and a special service in order to obtain information about specific resource. The information about the resources include: user-data specified when the instance is configured at start-up, password-less SSH public key, username and password pairs, network information.

VM resource registration is handled by the naming service which generates unique name randomly and registers it with the DNS. There are other services such as user-data service, package daemon and logging service which are responsible for user scripts, package installation and event logging. The implementation of mOS v1.X using openSUSE 13.1 uses the default ramdisk for boot with slight modifications in order to satisfies some requirements by the MODAClouds platform.

8.2.2 Platform Sub-systems

The run-time bootstrapper coordinates the deployment of the core packages as well as the supporting service packages. This is achieved by delegating most of the jobs to other subsystem. It serves as a kind of frontend for the operator and the service

deployment. It delegates most task to the resource allocator, node bootstrapper and controller, service deployer and finally the application deployer.

All of the above mentioned systems are crucial to the runtime. However, the main focus of this chapter is to detail the importance of supporting service for the MODAClouds runtime. Keeping this in mind, only some of the components used in the deployment of the supporting service are highlighted here. For example, the **node bootstrapper** is in charge of the initial mOS customization for the MODAClouds run-time environment. It runs as a local OS service, started at boot time or run time. It also applies all customization needed to start the runtime environment. The **node controller** is responsible with the management of the core services that runs mOS and supports the MODAClouds platform. It will start/stop and monitor the services to ensure that every main component of the execution platform is working as expected.

8.3 Supporting Services

The auto-discovery of services, previously mentioned in Sect. 8.2, depends to a large extent on the correct packaging and deployment of services. In order to run a service on the platform there are certain requirements that need to be met by the software. First, the software has to be packaged as an RPM which contains everything the service needs in order to run. These RPM packages can be made using the JSON based MODAClouds mOS Packager [5] or using the standard RPMSPEC for OpenSUSE 13 for x86_64. Any non standard dependencies must be provided together with RPM so that they can be published together in the MODAClouds repository. It is important to note that although specially designed for MODAClouds each supporting service is a standalone deployable tool outside the MODAClouds context.

In order to successfully deploy any service or component their runtime dependencies in term of other services must be specified. For example, the DDA (Deterministic Data Analyzer) tool depends at runtime on C-SPARQL. In addition, all TCP or UDP sockets on which the services listen must be specified. Finally, wrapper scripts are configuring through environmental variables the socket addresses on which services are allowed to listen and the remote service endpoints on which the service depends on.

The next subsections detail the most important supporting services from MODAClouds. These are integral for the correct functioning of the MODAClouds solution.

8.3.1 Object Store

The classic approach in software configuration is through configuration files which reside on the local disk, however such an approach is not very well suited for a Cloud environment, where VM's are started from identical templates (the VM images), and in most cases unattended, thus the configuration files must be rewritten at startup.

Luckily, for such a scenario, there are existing solutions, such as Puppet[1] or Chef.[2] However they also require a central database where the actual configuration parameters are stored. Moreover some of the deployed services might also want to store small state data, either for later retrieval, or for weak synchronization within a cluster. In this case the simplest solution is to use either a kind of database, or a distributed file system. This is the rational behind the development of the Object Store.

The Object Store provides an alternative to the more traditional locally stored configuration files. In the Object Store an object is a keyed container which aggregates various attributes that refer to the same subject. For example one could have an object to hold the configuration parameters of a given service (or class of services); or perhaps to hold the end-point (and other protocol parameters) where a given service can be contacted.

The object's attributes are: **data**, **indices**, **links**, **annotations**, and **attachments**.

A collection serves no other purpose than to group similar objects together, either based on purpose or type, or based on scope (such as all objects belonging to the same service). Collections can be used without being created first, and there is no option to destroy them (except removing one-by-one all the objects that belong to it). Therefore there are no other implications (in terms of performance or functionality) of placing an object in a collection or another, except perhaps easing operational procedures (such as removing all objects belonging to a service).

The most basic usage of an object would be to store some useful information, and have it available for later access. The stored data can be anything, from JSON or XML to a binary file, and besides the actual data it is characterized by a content-type. Later based on this declared content-type one can decide how to interpret the data. Although there can be a single data item for an object, one could easily use multipart/mixed to bundle together multiple data items; however it is advisable to avoid such a scenario and use either links or attachments.

Access to the data is atomic and concurrent updates are permitted without any locking or conflict resolution mechanisms, the latest update overriding previous ones, thus no isolation with lost-updates being possible. Although the data can be frequently accessed or updated without high overhead, it is advisable to cache operations by using the dedicated HTTP conditional requests. Because the data is stored temporarily in memory, it is advised to keep the amount of data small, well under a 100 kilobytes. Data that is larger should be handled as an attachment. In addition to its data, an object can be augmented with indices which enables efficiently selecting objects on other criteria than just the object key. An object can have multiple indices, each index being characterized by a label and a value, and it is allowed to have multiple indices with the same label.

The major difference between indices presented by this solution and other NoSQL or even SQL databases is that most other solutions build their indices based on the

[1] http://docs.puppetlabs.com/.

[2] http://docs.chef.io/.

actual data. In the case of the object store, the indices are built based on meta-data that is associated with the actual data (the indices attribute). By separating the indexing from the actual data we have greater control over how the data is stored and retrieved. We also optimize for those access patterns where the data changes frequently, but the values used by the indexer stay the same.

Links are the feature which allows an object to reference another one, building in essence a graph. For example one could have a service configuration object, holding specific parameter values, and pointing to a global configuration object, holding default parameter values. A link is characterized by a label and the referenced object key, and it is allowed to have multiple links with the same label or the same referenced object (therefore a many-to-many relation can be created). Unlike indices, links are scoped under the object, are unidirectional, and are not usable in selection criteria. Therefore one can not ascertain which objects reference a given target object (without performing a full scan of the store). The only operation, besides creation and destruction, that can be applied to a link is link-walking, where by starting from an object, one can specify a label and gain access to the referenced object's attributes; link-walking can be applied recursively. Links can be destroyed or created as frequently as necessary as they are not indexed.

Data that logically belongs to the object, but which is either too large to be used as actual data or is static, can be placed within an attachment. Attachments are created in two steps. First, the attachment is created by uploading its content, and obtaining its fingerprint, so if the same data is uploaded twice the fingerprint remains the same thus no extra storage space is consumed. Second, a reference to the attachment (i.e. its fingerprint) is placed within the object with a given label, together with the content-type and size which serves only for informative purposes. The same attachment can be referenced from multiple objects without uploading its data, provided that the fingerprint is known.

Similarly, accessing the attachment of an object is done in two steps: obtaining the attachment reference, then accessing the actual attachment based on its fingerprint. Like with links, attachments are scoped under an object, only their data being globally stored. In terms of efficiency, creating or updating attachments do not have high overhead (except the initial data upload). This is because the various information pertaining to a specific object such as the actual data, meta-data, links, annotations, attachments are not lumped together. These are partitioned, just like vertically partitioned SQL databases. Also, because attachments are identified based on their global qualifier, duplicating or moving an attachments from one object to another doesn't require the re-upload of the entire attachment.

The annotations are meta-data which can be specified for objects or attachments, and are characterized by a label (unique within the same object) and any JSON term as a value. Annotations are those data which if erased do not impact the usage of the object. In general annotations can be used to store ancillary data about the object, especially those used by operational tools. For example, one can specify the creator, tool and possibly the source, ACL's or digital signatures, etc.

The object store has facilities for multi-Cloud deployment via replication. The replication process has three phases: defining on the target (i.e. the server) a replication stream, which yields a token used to authenticate the replication; defining on the initiator (i.e. the client) a matching replication stream; and the actual replication which happens behind the scenes. It must be noted that the replication is one way, namely the target (i.e. the server) continuously streams updates towards the initiator (i.e. the client). If two-way replication is desired, the same process must be followed on both sides.

Regarding conflicts, and because internally the object store exchanges "patches" which only highlight the changes, any conflicting patch is currently ignored. It is therefore highly recommended to confine updates to a certain object only to one of the two replicas. However if multiple changes happen to the same object, and multiple patches are sent, and say the first one yields a conflict, but the rest don't, only the conflicting patch will be discarded, the others being applied. It is possible to obtain replication graphs or trees, including cycles, and the object store handles these properly.

Service Configuration Use Cases

Let us suppose that an operator has several instances of the same service type (i.e. application server or database) which he would like to configure during execution. Moreover the user would like to change the configuration and have it dynamically applied as easily as possible.

Single shared configuration is the most basic scenario. The most simple solution is to store the configuration parameters in an object created before execution is started, preferably JSON term or plain text as the data, or alternatively as an attachment. Then at execution the object's reference is specified as an initialization argument to each of the instantiated services, which retrieve the data and use it to properly configure the service.

If each service continuously polls the object for updates, it can detect when the operator has changed the configuration parameters, and apply the necessary changes (possibly by restarting). This might seem to involve fetching the data over and over again thus incurring large network overhead, such is not necessary true if one uses HTTP conditional requests which is rather efficient.

In the case of **Multiple shared configurations** the services require multiple different configuration parameters grouped in multiple "files", possibly because their syntax is different, or perhaps for better maintenance by the operator. One solution to this problem is to create a master object and using links to point to the other needed configuration objects. As before polling can be applied to detect configuration changes, but because now it involves multiple objects, after an update has been detected a grace period should be used, after which another check should be done, if no other updates have been detected the configurations are applied. This prevents frequently restarting the service while the operator updates sequentially the configuration objects.

8.3.2 Artifact Repository

The artifact repository is designed as archive of artifacts generated in various parts of the MODAClouds Project. The project aims to be able to store information like deployment recipes, maven artifacts, software packages or, basically, any other data. The Artifact repository provides an API for managing the artifacts and for searching the stored data based on their meta-data. The API is REST [1] compliant, and consumable from all MODAClouds components and development tools.

It has to satisfy a set of fairly simple requirements. It has to enable the upload of binary files (BLOB). An artifact may be composed of on or more files under 1 GB. Each artifact has to be versioned as any modification done to an existing artifact has to be identifiable. Also, each file associated with an artifact has to be downloadable and it has to support a number of repositories.

The artifact are stored directly on the file system. The file hierarchy is directly mirrored from the URL structure. This means that the folder structure will include folders for repositories, artifacts, versions and the files. Thus making interrogation extremely intuitive. Another bonus of using a simple file system based approach is the ability to use rsync as the synchronization mechanism between artifact repository deployments. In some ways it can be considered as a stripped down version of the object store. It's main design goal was to create a simple yet powerful mechanism to store software artifact can handle much larger files than the object store.

8.3.3 Load Balancer Controller

The goal of the load balancer controller, is to provide a RESTful API that is able to control and configure Haproxy.[3] For this we used a micro-framework written in python called flask.[4] It is designed as an extensible framework with little dependencies. The main two dependencies are the web server gateway interface subsystem represented by Werkzeug and Jinja2 which provides template support. It is important to note that flask does not natively support some required functionalities such as accessing databases, however there are a significant number of extensions to the framework that resolve these shortcomings [2]. During the project we developed a python Haproxy RESTful API (modaclouds-loadbalancer-controller or MLC) which based on the users input generates a configuration file for the load-balancer (Haproxy) thus controlling its behavior. It exposes both frontend and backend settings as well as limited support for ACL specifications. At this point it is important to note that MLC doesn't check if the ACL triggers are correct when first entered by the operator.

It stores all interactions in a sqlite database, which also serves as the basis of the configuration file. The jinja2 template engine is used to generate the configuration

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

[4] http://flask.pocoo.org/docs/0.10/.

file which is then loaded into Haproxy. Currently each configuration file is saved into the database and can be accessed by querying the database. The API is designed to:

- **add, edit and delete resources**—This means that pools, gateways, endpoints and targets can be defined. These represent direct representations of resources present in Haproxy. Each interaction is saved and versioned.
- **set policy**—Load-balancing policies and their associated parameters can be set of each target. For example in the case of round-robin we can set the weights for each target.
- **start Haproxy service**—First a configuration file is generated and used to start the load-balancing service. Each time a new configuration is generated it is reloaded into the already running service.

The MLC is designed to hide as much technical details of Haproxy as possible. This is done in order to make the REST API as agnostic as possible. For example in MLC we use the term gateway to define a frontend server and pool to define the backend servers. This enables easy extension of the MLC and the REST resource structure can be easily mapped onto other load-balancer solutions (such as ngnix) besides Haproxy.

8.3.4 Batch Engine

The main goal of the Batch Engine (BE) is to support the computationally-intensive routines that are to be executed as part of the Filling the Gap Analysis. As there are no tight deadlines, these routines are executed offline, and therefore it is possible to exploit the large datasets of monitoring information collected at runtime. We therefore opt for a BE that exploits a pool of parallel resources. In particular, the BE aims to provide on demand HTC/HPC clusters on top of existing computational Cloud resources (e.g., Eucalyptus, EC2, Flexiant, PTC, etc.).

From a technical perspective, the BE integrates the services provided by the underlying scheduling middleware, particularly the HTCondor workload management system [3]. The BE provides REST API's that allow job execution management (including submission and monitoring).

The API offered by BE is extensible, providing the ability to support new job-scheduling engines or middleware. As the FG analysis techniques were implemented in Matlab, we are making use of the Parallel Toolbox and the APIs offered by the BE to submit and manage the parallel jobs, as well as to retrieve the results. The execution of the FG analysis relies on the Matlab Compiler Runtime (MCR), a free runtime platform to execute standalone applications developed in Matlab.

The main features of the BE are include automatic provisioning using specially-designed Puppet modules, the ability to use existing infrastructure (ex: Amazon EC2, Flexiant) and an API middleware for job control. There are several important features in the BE. First, a REST API (based on JSONRPC2) for controlling the deployment.

This API allows to dynamically specify the architecture of the provisioned cluster, and to reuse predefined models. It allows customizing the cluster based on the required resources (CPU, memory, GPUs, etc.). This API abstracts the cluster deployment operations, including: machine deployment; software provisioning, configuration, monitoring. The API resorts to specially-defined Puppet modules that handle the deployment of all the software components.

It also uses a REST API for job management and monitoring. This REST API abstracts the job management operations and interacts with the back-end HTCondor service. The API provides common operations offered by HTCondor as REST-compliant services. These operations include job submission, data staging, job state notifications, etc.

Lastly a flexible core that allows the addition of various schedulers, each with a different feature set, as required by applications.

From an architectural point of view the BE is composed of four main subsystems: **Batch Engine API**: This subsystem is responsible for interacting with the client applications or users. It handles the requests and delegates them to the other subsystems.

Batch Engine Cluster Manager API: Based on SCT,it uses the Configuration Management subsystem (mainly Puppet) and the Cloud interface for deploying nodes and provisioning the job scheduler (e.g., HTCondor).

Batch Engine Execution Manager: Is responsible with the effective job execution and corresponding event handling (interaction with external components). It dispatches job execution requests to the deployed HTCondor workload manager. The workload manager permits the management of both serial and parallel jobs, feature that will be exploited by applications that use MPI like technologies.

Scheduler: Represents the effective job-scheduling system, responsible for executing the submitted jobs. It also provides the wrapping mechanism needed for offering integration facilities like the job notification API.

Finally, the FG Analyzer calls the Batch Engine periodically and executes several jobs on multiple nodes performing different analyses. For instance, the FG Analyzer can execute several demand estimation procedures in parallel using the Batch Engine to compare the accuracy of them during design time. It also executes the analysis corresponding to different datasets in parallel, thus speeding up the analysis phase.

8.4 Conclusions

As we saw in the previous sections, there are a wide array of tools and platforms that make up the complete MODAClouds solution. The MODAClouds platform core components are comprised of more than 70 RPM packages. Some of these packages are custom repackages of components such as the Java Virtual Machine, Go runtime, python interpreter, Haproxy etc. These are packages on which MODAClouds platform services depend upon. For the sake of completeness we will list the components that comprise each MODAClouds platform:

- **Creator 4Clouds**—Filling the Gap (FG) Analyzier, Functional Modelling Tool, Space 4Cloud, LINE, CloudML, DATA Mapping
- **Venues 4Clouds**—Decision Support System
- **Tower 4Clouds**—Monitoring Manager, DDA, Data Collector, QoS Models, Metrics Observer, Metrics Explorer, Knowledge Base, Matlab SDA, Weka SDA
- **SpaceOps 4Clouds**—Self-Adaptation Stress tester, Load-Balancer reasoner
- **Energizer 4Clouds**—Load Balancing Controller, Object Store, Artifact Repository, Data Migration, mOS image, mOS package builder

Most components from Tower 4Clouds [7], SpaceOps 4Clouds [6] and Energizer4 Clouds [5] are packaged and deployed on top of mOS. Even more significant is the fact that most tools use the supporting services in order to fulfill their function. For example the Load Balancer Reasoner uses the Load Balancer controller supporting service in order to adjust the weights of server backends in Haproxy. Without this REST interface based controller the reasoner would not be able to function. This controller is also used by the Models@Runtime component and can be used by any other application that needs a load balancer. Similarly the object store and artifact repository are used by the Tower 4Clouds components and Load Balancer reasoner, while the Batch engine is used by the Filling the Gap tools.

This chapter has provided an overview of the deployment and architecture of the supporting services and runtime platform. It has highlighted the importance of these types of services which play an important role in the MODAClouds Runtime platform (Energizer 4Cloud). We have also covered how services from Tower 4Clouds and SpaceOps 4Clouds are packaged and later deployed on top of mOS. We have also described the fact that each supporting service is a self contained software package meaning that they can be easily reused and modified. The four supporting services, Object Store, Artifact Repository, Load-Balancer Controller and Batch Engine have been described and rationale behind their design has been covered. Lastly these services are put into context of the MODAClouds runtime platform (with details how each supporting service is integrated into sub-system of the runtime platform).

References

1. Fielding RT, Taylor RN (2002) Principled design of the modern web architecture. ACM Trans Internet Technol 2(2) ISSN 1533-5399
2. Grinberg M (2014) Flask web development: developing web applications with python. O'Reilly Media Inc. ISBN 1449372627
3. Thain D, Tannenbaum T, Livny M (2005) Distributed computing in practice: the condor experience. Concurr Pract Exp 17
4. Petcu D, Macariu G, Panica S, Crăciun (2013) Portable cloud applications—from theory to practice. Future Gener Comput Syst 29. ISSN 0167-739X
5. Iuhasz G, Panica S, Casale G, Wang W, Jamshidi P, Ardagna D, Ciavotta M, Whigham D, Ferry N, González R (2015) MODAClouds D6.5.3—runtime environment final release. http://www.modaclouds.eu

6. Fortiş F, Iuhasz G, Neagul M, Casale G, Perez J, Wang W (2015) MODAClouds D5.3.2—techniques for filling the gap between design time and runtime. http://www.modaclouds.eu
7. Casale G, Weikun W, Miglierina M, Munteanu V (2014) MODAClouds D6.3.2—monitoring platform final release. http://www.modaclouds.eu

Open Access This chapter is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, duplication, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the work's Creative Commons license, unless indicated otherwise in the credit line; if such material is not included in the work's Creative Commons license and the respective action is not permitted by statutory regulation, users will need to obtain permission from the license holder to duplicate, adapt or reproduce the material.