POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica

Using Models at Runtime to Support Adaptable Monitoring of Multi-Clouds Applications

Dipartimento di Elettronica Informazione e Bioingegneria

Relatore: Prof. Elisabetta Di Nitto
Correlatore: Ing. Marco Miglierina
Correlatore: Dr. Nicolas Ferry

Tesi di Laurea di:
Lorenzo Cianciaruso, matricola 799375
Francesco di Forenza, matricola 799679

Anno Accademico 2013-2014

Alle nostre famiglie.

Abstract

Cloud Computing has gained a lot of popularity in the past years by offering easy access to a shared and virtualized pool of computing capabilities, emphasizing flexibility, scalability, and performance of applications. The ability to run and manage multi-clouds applications (*i.e.,* application that runs on multiple public or private clouds) allows exploiting the peculiarities of each cloud solution and hence improves non-functional aspects such as availability, cost, and scalability. Monitoring such multi-clouds applications is fundamental to track the health of the applications and of their underlying infrastructures as well as to decide when and how to adapt their behaviour and deployment.

It is clear that, not only the application but also the corresponding monitoring infrastructure should dynamically adapt in order to be optimized to the application context (*e.g.,* adapting the frequency of monitoring to reduce network load) and to enable the co-evolution of the monitoring platform together with the multi-cloud application (*e.g.,* if a service migrates from one provider to another, the monitoring activities have to be adapted accordingly).

The goal of this work is to extend a model-based platform for the dynamic provisioning, deployment, and monitoring of multi-clouds applications to enable the dynamically adaptation of the monitoring activity that best fit with the actual deployment of the application. The model representation of the deployment simplifies the reasoning on the system and eliminates the separation between design and run time.

The results achieved permit to co-evolve the monitoring activity together with the status of the running system, from the initial deployment to further modifications of the multi-cloud application. In this way, the monitoring component can autonomously monitor new resources belonging to different cloud providers and stop monitoring resources that are no longer deployed.

Sommario

La popolarità del cloud computing è cresciuta esponenzialmente negli ultimi anni a causa della facilità con cui permette di accedere a risorse computazionali virtuali, enfatizzando in tal modo la flessibilità, la scalabilità e le prestazioni delle applicazioni. La possibilità di eseguire e gestire applicazioni multi-cloud (cioè applicazioni eseguite contemporaneamente su cloud diversi, pubblici o privati) permette di sfruttare le caratteristiche migliori di ogni servizio, accentuando ancora di più i vantaggi tipici del cloud computing stesso: affidabilità, riduzione costi e scalabilità. Il monitoraggio di queste applicazioni multi-cloud assume un ruolo di rilievo per determinare lo stato di salute delle applicazioni stesse e dell'infrastruttura sottostante. Al contempo, questa attività di monitoraggio è necessaria per decidere quando e come adattare il comportamento delle applicazioni o l'infrastruttura che le ospita.

Risulta evidente come la stessa piattaforma di monitoraggio debba, a sua volta, essere adattabile per ottimizzare l'attività di monitoraggio (es. ridurre la frequenza di funzionamento per liberare risorse) e per poter coevolvere con l'applicazione (es. seguire automaticamente l'applicazione su un cloud differente).

Lo scopo di questo lavoro è estendere un sistema basato su modelli per creare, gestire e monitorare applicazioni multi-cloud, permettendo che l'attività di monitoraggio possa essere automaticamente e dinamicamente ottimizzata in base allo stato dell'applicazione. Costruire una piattaforma basata su modelli permette inoltre di semplificare il lavoro di entità esterne che devono interagire col sistema. In aggiunta, un modello continuamente sincronizzato col sistema permette di interagire su di esso anche durante l'esecuzione rompendo, di fatto, la separazione tra modellazione e mantenimento dell'infrastruttura.

La piattaforma realizzata permette la coevoluzione dell'attività di monitoraggio e dell'applicazione dal momento del primo rilascio a tutte le modifiche successive, abilitando l'autoadattamento della stessa attività di monitoraggio.

Acknowledgements

It is a pleasure to thank those who made this thesis possible with advices, critics and observations.
We would like to thank our supervisor Prof. Di Nitto and our two mentors Eng. Miglierina and Dr. Ferry: without their help and support, this thesis would not have been possible.

We would like to thank Dr. Solberg who kindly let us developing part of this work at Sintef and all the colleagues who have greeted and helped us during the three months in Norway.

A special thanks to the students met in Sintef with whom we spent a nice and pleasant period in Oslo.

A profound thanks to Debora for the affection and care shown during the study and especially in life.

We owe our deepest gratitude to our families and friends for the continuous support during these years at university.

Finally, we would like to thank each other for having lived together this experience.

Contents

List of Figures

Listings

Chapter 1

Introduction

"If education is always to be conceived along the same antiquated lines of a mere transmission of knowledge, there is little to be hoped from it in the bettering of man's future
For what is the use of transmitting knowledge if the individual's total development lags behind?"

Maria Montessori

1.1 General context

Cloud Computing has gained a lot of popularity in the past years by offering easy access to a shared and virtualized pool of computing capabilities [1], emphasizing flexibility, scalability, and performance. The ability to run and manage multi-clouds applications [2] (*i.e.,* application that runs on multiple public and private clouds) allows exploiting the peculiarities of each cloud solution and hence improves non-functional aspects such as availability, cost reduction, and scalability.

The multi-cloud gives the possibility to get all the advantages that the Cloud Computing offers and in addition, it prevents the vendor lock-in. Avoiding the vendor lock-in the developers are no more bounded up with a single cloud provider, but can choose among multiple providers during the life-cycle of the cloud applications. Moreover, they can exploit the key success factors of each one to obtain the best solution that fits with the requirements.

By the way, there are some disadvantages: dealing with a multitude of cloud providers makes the design of a multi-cloud application more complex. Also the management of multi-cloud applications presents more difficulties: each cloud provider has its own terminology, APIs, and interfaces and the way to approach to each one is different.

Nowadays there are several solutions to support the design of multi-cloud applications, but they are not sufficient to manage properly the complexity of the development and the administration phases [3]. The purpose of such tools is especially the collection of data about an application in order to analyse its behaviour and state of health.

Monitoring applications and their underlying infrastructure is fundamental to study the performances, understanding for instance if the application is overloaded or if the response time is limited, and then decide when and how the application should be adapted to satisfy the non-functional requirements.

Adapting an application allows to fulfil specific requirements according to the runtime status of the deployment, also improving the user-experience of the final user. A key challenge when performing such adaptations consists in modifying accordingly the monitoring activity whilst limiting the overhead it induces. Once that the adaptation has been completed also the new resources should be automatically monitored.

The MODAClouds project in which we are involved proposes to take care about these issues in the multi-cloud context. It is a very ambitious work that aims to completely automate the deployment and to manage the entire life-cycle of a multi-cloud application. Moreover, it offers an IDE to interact with the user and to show information about the system. In order to manage the life-cycle of an application, the MODAClouds architecture contains a monitoring platform that collects information from the resources deployed, a reasoning engine that takes decision about adaptations to perform on the deployment based on the data received in input, and a deployer to act on the running system.

What was missing before our contribution was a communication channel between the component that manages the deployment at runtime, and the components that monitors the resources in the meantime. Due to this communication lack, the monitoring platform could not evolve and monitor the updated system. Starting from the initial deployment of the application, passing to further updates of the same, the monitoring could not start without information about the deployment

(*e.g.,* which resources compose the application and how they are related).

To remedy to this problem our goal was to create a communication channel between the two components, permitting them to exchange information about what is occurring at run-time in the system.

The creation of the communication channel allows the automation and adaptation of the monitoring activity according to the real situation, completely avoiding user's interventions. In this way, the user would be able to analyse the data presented by the MODAClouds Monitoring Web Interface, because the monitoring activity will be correctly performed although there have been adaptations dictated by the reasoning engine.

1.2 Brief description of the work

As already written in the previous section, cloud computing is spreading all over the world, and several tools permit the management of cloud applications. The tools that we are now going to briefly present and that we will describe in detail in Chapter 2, are strictly related to our work. Nevertheless, there are some differences: some solutions are proposed in a single cloud boundary and others do not support adaptation.

Currently some tools, such as JadeCloud [4], allow managing and monitoring of both cloud infrastructure and software resources. Through the extension of other services (detailed in Chapter 2), JadeCloud offers the data in a unique interface giving the possibility to the users to analyse them autonomously. It offers a dashboard to the end user that permits to observe the status of the system and applications, but not to automatically perform some kind of adaptation. As well, Brooklyn is a framework for modelling, monitoring, and managing applications through autonomic blueprints [5]. However, no support is given for adapting Brooklyn monitoring platform based on performance issues.

Related to the multi-cloud environment there is a tool called JCatascopia [6] focused on the multi-cloud monitoring. It can retrieve heterogeneous information both at machine level (*e.g.,* CPU and Disk usage) and at application level (*e.g.,* throughput, latency, and availability). It also offers a rule mechanism permitting to the developers to aggregate and activate new metrics.

All these tools are examples focused on the monitoring of an application, but none has the ability to auto-adapt the monitoring in case of variation of the deployment, because they do not know the whole structure of the deployment model.

This is the reason why our concept is to build a system that is model driven, in the sense that the adaptation of the monitoring activity is based on the model of the status of the deployment. Model-Driven Engineering [7] is a branch of software engineering, which aims at improving the productivity, quality, and cost-effectiveness of software development by shifting the paradigm from code-centric to model-centric. This approach, which is commonly summarised as “model once, generate anywhere”, is particularly relevant to face the complexity of developing complex systems such as multi-cloud systems [8]. Using a model-driven system we aimed to realize a platform capable of dynamically adapt itself according to the modifications of the environment. These Dynamically Adaptive Systems (DAS),

which enable the continuous design and adaptation of complex software systems, typically rely on an architecture inspired by control-loops [16]. This architecture is composed of three main activities: *Sense* to collect information that reflects the current state of the system, *Plan* "to reason" and to decide how to adapt accordingly, and finally *Act* to enact the decided adaptation.

Our work is in the context of the MODAClouds project. The main task is to integrate the Monitoring Platform provided by Politecnico di Milano, with the Models@Runtime engine developed by Sintef (Oslo, Norway), both involved in the MODAClouds project. The Monitoring Platform handles the monitoring activity on the multi-cloud application. It is responsible for collecting metrics about the running system. To start the monitoring activity, the Monitoring Platform needs information about the resources in the running system, so it contains its own model where all the components to monitored are detailed. On the other hand, the Models@Runtime engine manages the deployment of the applications. It consists in enacting the provisioning, deployment, and adaptation of multi-clouds systems. In addition, the Models@Runtime engine provides an abstract and up-to-date representation of the underlying running system. A change in the running system is reflected automatically in the model of the current deployment. Similarly, a modification to this model is enacted on the running system on-demand. This causal connection enables the continuous evolution of the system with no strict boundaries between design-time and run-time activities.

Considering the continuous evolution of the system, the goal of our work is to keep the also model in the Monitoring Platform up-to-date and consistent with the status of the application. A coherent model in the Monitoring Platform permits to adapt the monitoring activity in the right way. Moreover, there are also two other targets: one is to enable the communication also in the opposite direction, in order to enrich the representation of the deployment in the Models@Runtime engine with some monitored data coming from the Monitoring Platform. The second one is to give to the Models@Runtime engine also the ability of discovering the status of the virtual machines composing the running system.

Keeping the model in the Monitoring Platform up-to-date also enables the possibility to make adaptations. In the Monitoring Platform are possible two kinds of adaptation: the parametric adaptation and the structural adaptation. The parametric adaptation changes the way in which data are collected from the resources. The Monitoring Platform permits to install some monitoring rules to specify which data

should be collected and in which way. Though the monitoring rules, it is possible to change the frequency of the data acquisition and the sampling probability. There is also the possibility to trigger an action when a particular condition, on a particular data flow, is violated. The parametric adaptation allows modifying the workload of the Monitoring Platform, lowering the impact of the monitoring activity, or retrieving more information from the machines in case of need.

Our work is focused principally on the structural adaptation. The structural adaptation concerns the adaptation of the Monitoring Platform to follow the structural modifications in the deployment of the application. The Monitoring Platform has to be able to adapt itself to the new configuration of the running system. The structural adaptation permits to start and stop automatically the monitoring activity on the resources that have been instantiated or de-instantiated.

The only way to ensure that both kinds of adaptation are performed correctly is to have the model inside the Monitoring Platform consistent with the status of the running system. The model should contain not only the machines, the applications, and the methods but also the relationships among them. This model, indeed, is used by the Monitoring Platform to interpret the rules and decide how to perform the monitoring activity and should be updated and consistent (*e.g.,* if a machine is not in the model or the data about it are wrong it cannot be monitored because the Monitoring Platform does not know how contact it). Our contribution was focused precisely on finding a way to keep the model up-to-date after some changes happen in the running system.

Our target is to offer a platform different from the current products and capable of achieving the described goals, but at the same time easy to update and to manage. In order to achieve this result, we identified the following requirements used through the entire document to drive our design and implementation choices:

- **Cloud provider-independence (R_1):** our platform should support a cloud provider-agnostic specification of the monitoring, provisioning, and deployment of multi-clouds applications. This simplifies the design and the adaptation of both the multi-clouds application and the monitoring platform, and prevents vendor lock-in.
- **Model abstraction and coherence (R_2):** our platform should provide an up-to-date, abstract representation of the running system. This will facilitate reasoning, simulation, and validation of adaptation actions before their enactment.

- **Co-evolution of the monitoring model with the deployment (R_3):** our platform should maintain the monitoring model consistent with the actual deployment of the running system. This will avoid the user to reconfigure the Monitoring Platform's model after changes in the deployment.
- **Adaptation of the monitoring platform at run-time (R_4):** our platform should provide support for dynamic adaptation of the Monitoring Platform's processes. This will permit to modify the monitoring processes at run-time.

At the end of the implementation, we tested and experimented the correct way of work of the system deploying and managing a multi-cloud application. The first target was to deploy a multi-cloud application with the components offered by the MODAClouds project. Once deployed, the focus moved to the information sent to the monitoring activity in order to start monitoring. After the verification of the correctness and the coherence between the models, we tested if the evolution of the running system is handled correctly both from deployer side and from monitoring side. During these operations, we verified that the monitoring component was collecting the data correctly and that it reacted promptly and properly to the adaptations of the deployment. The experiment was conducted in a multi-cloud environment with successful result, fulfilling the requirement of cloud provider independence (R_1). The Models@Runtime engine reflected properly in its own model the status of the running system, considering also the status of the virtual machines declared by the cloud provider (R_2). The system was capable of installing and maintaining the correct model through an efficient exchange of information from the Models@Runtime engine, which advised the Monitoring Platform about any adaptation every time it occurred (R_3). On the other side, the monitoring components properly handled every update respecting all the dependencies among the components. Consequently, the monitoring activity was consistent with the status of the deployment in every moment (R_4).

1.3 Outline

In this chapter, we have given the general context and the general goals of the work.

In Chapter 2 is described the actual state of the art in the context of our work, showing the solutions present on the market that are related to our work. It presents tools supporting the management and the monitoring of applications in both cloud and multi-cloud environments.

Chapter 3 initially describes the architecture of the system, presenting in detail each component involved. After this, the problems encountered with the current tools are described, providing motivating examples that concretely show the problems in the actual scenario. From the motivating examples are then extracted the requirements that the solution must satisfy to successfully achieve the target. The same examples will be used again in the other chapters to show how our solution permits to overcome the problems.

Chapter 4 proposes a solution to get rid of the problems described in Chapter 3. This chapter contains the motivation of all the design decisions taken in order to integrate the various components to achieve the solution.

Chapter 5 shows how the designed choices have been technically implemented, indicating the patterns and the technologies used, explaining how the model integration between the Monitoring Platform and Models@Runtime engine is performed, and describing the communication paradigm adopted to exchange information.

Chapter 6, taking the cue from the motivating examples in Chapter 3, illustrates how the solution permits to solve the problems and offers a short demo of usage of the solution proposed to highlight the performances and the results of the work.

Finally, Chapter 7 draws the conclusion, recaps the results obtained with respect to the requirements and suggests some activities that can be done as extensions of this work.

Chapter 2

State of the art

"Computer science is no more about computers than astronomy is about telescopes."

Edsger Dijkstra

In this chapter, we present the state of the art related to our work focusing the attention on platforms that support the monitoring and the management of cloud applications, software products exploiting features of the cloud environment, tools that simplify the user experience, and research papers arguing with similar problems.

The context of our research is strictly the multi-cloud environment (*i.e.,* our platform is also capable of working with a single-cloud application but we cannot make a priori the hypothesis that user will deploy a single-cloud application) but this does not exclude that there are similarities between our work and single-cloud instruments. Moreover, the most advanced solutions in terms of functionalities, stabilities, and performances are the ones offered directly by cloud providers so intrinsically single-cloud. Therefore, we present also these tools to have a clear overview and to analyse if our work offers functionalities that can be compared with the technologies that are actually used.

To simplify the reading we divide the related works into two main groups: the first one focused on managing application running on a single cloud (*i.e.,* only one provider) and the second one containing tools capable of managing multiple clouds simultaneously.

2.1 Single cloud

2.1.1 Provider level

One of the major goals of outsourcing servers from a company is to simplify the management and the set up of the servers. For this reason, the cloud providers try to offer tools to manage virtual machines and to analyse data coming from their cloud. The providers compete among themselves to offer powerful interfaces to manage in few clicks a large number of machines and to monitor them.

If we analyse the biggest companies offering cloud services, we observe that the functionalities of their software are similar and all of them continue copying features from each other.

The fundamental tools that more or less all the competitors offer are three: a console, a performance monitor and an alarm service. The console, often in the form of a web application, is used to offer the basic functionalities such starting, stopping, and deleting a virtual machine, managing security groups and billings. The performance monitor shows information about the status of the various resources (*e.g.,* CPU usage, number I/O requests...). The alarm service works in deep connection with the performance monitor and gives to the user the possibility to trigger an alarm (*e.g.,* mail, and SMS) if certain conditions occur. Some providers also offer the possibility of specifying some operations to execute in reaction to an alarm (*e.g.,* scale up, scale down a machine).

Amazon

Currently Amazon is one of the most powerful competitors among cloud providers. It offers large-scale cloud computing resources through the Amazon Web Service [9] platform (*i.e.,* AWS). In Amazon, the user can access to the AWS console in order to create and manage virtual machines. Examples of operations that can be performed using the console are to spawn or destroy a machine, manage security information, load a specific OS on a machine, create a back-up image... To check the status of the virtual machines it offers Cloudwatch [10], a proprietary monitoring platform that collects data from the running machines and shows them graphically at different levels of aggregation. This data can be saved and analysed with more advanced tools to extract hidden information or obtain data that can be used to take strategic and long-term decisions. An example is AWS Elastic

Beanstalk [11], a service for deploying and scaling web applications and services developed with Java, .NET, PHP, Node.js, Python, Ruby, and Docker on familiar servers such as Apache, Nginx, Passenger, and IIS. Elastic Beanstalk automatically handles the deployment providing: capacity provisioning, load balancing, and auto-scaling.

The offering of tools is completed by a trigger system, SNS [12], able to send notifications, using mail and SMS in case some user-defined conditions occur.

Since all these services are accessible also from APIs, it is possible for the user to create ad hoc tools to integrate the functionalities offered by Amazon with external instruments having total control on the resources.

In addition to these services, Amazon offers many tools not strictly related with the provisioning and the monitoring of the resources, but useful to manage a large and complex cloud infrastructure. There are instruments to analyse the billings and automatically identify some points that can be improved to achieve better performance and/or cost reduction [13]. Another important set of tools is focused on computer security, detecting intrusions and finding weak points in the infrastructure. The problem of this ecosystem is that it works only on AW2 machines: machines directly hosted by Amazon. A limited integration with other systems (*e.g.,* private cloud) is possible but, for evident reasons, is not possible to exploit the functionalities offered by AWS services on different providers, closing de-facto the door to multi-cloud applications.

Rackspace

Rackspace, in a similar way as AWS, offers its own cloud computing-environment. They claim to offer much more than infrastructure alone, since they deem that high-performance and a reliable infrastructure is not enough to succeed in the cloud. For this reason together with their tools, Rackspace offers a team of experts to support the user. In addition to the web application that offers the common functionalities, there is Rackspace Monitor [14].

According to its description, Rackspace Monitor is a set of monitoring tools to "continuously monitor the entire infrastructure stack at a customizable level of detail that pinpoints core issues for speedy resolution". It is an enterprise-grade solution designed to keep applications up and running fast all of the time. It includes remote connectivity tests from regional zones and agent-based monitoring to gather information from the inside of each resource. This service is integrated

deeply with the underlying systems and supports the process of identifying and solving the problems that may arise at run-time. Among all the functionalities offered by Rackspace Monitor there is an Alarm and Notification service to send real-time notification in case the user attention is required.

Flexiant

All the analysed providers offer the basic tools to perform the operation described above with more or less attention to some specific points. We would like to cite also Flexiant [15] a relatively small cloud provider based in London that is partner in the MODAClouds project. Through Flexiant Orchestrator, they offer all the common features to deploy and manage a cloud environment and Flexiant web application includes also a basic tools to monitor the machines status.

In addition to these "standard" features, Flexiant is trying to push to limit the notification services offering a way to undertake actions automatically if some conditions are verified [16]. Through in-nested triggers it is possible to create, directly on the Flexiant console, complex rules to react to particular situations. This innovative technology opens a door on the possibility to auto-adapt the deployment according to the data coming from the monitoring activity that is one of the goals of our project.

All the tools we analysed above are easy to use and well integrated with the underlying system but they are proprietary and platform dependent. For this reason, they cannot be used in a multi-cloud environment. Moreover, the tools are provider specific and, even if there is no need to use a multiple cloud environment, there is no easy way to migrate the entire system to another provider. The goal of the software offered by providers is the customer retention, making difficult to migrate to another provider. These aspects make this approach very different from the one proposed in our work, since we try to create a level of abstraction to hide the actual service provider. Providing a level of abstraction and then deal separately with the various services is also a way to break the connection between the user and the cloud companies, offering a higher-level system in which the actual service provider only offers mere computational power.

2.1.2 Application level

A big limitation of the tools presented is that they are focused only on the monitoring of the machines and they do not deal with the running applications. Even if the status of the application can be inferred from the status of the machine (*e.g.,* if the CPU utilization falls to zero, it is probably because the application is no longer working and may be in error), it is insufficient in almost all the contexts. To solve the described problem there are other works that developed tools to manage and monitor both cloud infrastructure and software resources in glance. This is something more than just monitoring the bare machines and can be useful in many real cases. The goal of these programs is to offer a unique way to collect data from both applications and virtual machines.

JadeCloud

JadeCloud [4] is a tool designed and partially implemented that aims to extend the services provided by Openstack [17] and Hyperic [18]. The goal of this tool is to integrate the data coming from both the machines and the applications, presenting them to the user in a unique interface.

This work has various common aspects with our targets. JadeCloud uses models@runtime pattern [19, 20] to manage the running system and it enriches the representation of the system with data from the provider and the deployed applications. The main differences are that: their work is focused only on providers offering Openstack compatible systems and they do not perform adaptation on monitoring service.

Google App Engine

Google App Engine [21] offers a way to manage cloud application, not at IaaS (*i.e.,* Infrastructure as a Service) level, but at a PaaS (*i.e.,* Platform as a Service) level. Using Google App Engine it is possible to write and run applications in an environment that offers many facilities. There is the possibility to focus the attention on the application using the storage services, the network communication towards high-level calls offered directly by the platform.

The problem of working in similar environment is that the code has a strong dependency from the platform and the offered services, making the migration to a

different provider expensive or almost impossible. In other words using a PaaS is very easy to be stuck by the vendor lock-in.

Microsoft Azure

Another reality in the cloud providers is Microsoft with the new platform Azure [22]. Azure offers both PaaS and IaaS services. It proposes many Microsoft tools to manage DBs or network services but is possible to integrate also products of other vendors.

It offers all the instruments to create and manage the creation of a virtual environment. In addition Microsoft Azure offers a "Management Portal" from which is possible to access and customize the monitoring metrics. They stress a lot the way in which the metrics can be aggregated and plotted in various ways to monitor easily the Key Performance Indicators of a system. In addition, all the monitoring data are stored in a storage account, which can be accessed even outside of the portal.

Azure offers also the possibility to create rules to automatically send an email to administrators if the value of a metric reaches a given threshold. Part of these functionalities is integrated in Visual Studio so is possible to plan and organise the monitoring activity and to specify the monitoring components during development phase and not only at run-time.

2.2 Multiple clouds

Moving to the multi-cloud approach, some studies [23] have been done about possible frameworks to simplify the management of multi-cloud applications.

The tools in this section consider the close relations between Clouds and applications, trying to perform monitoring across all layers. Multi-cloud deployment further complicates the monitoring activity due to lack of cross-platform support for making the monitoring solutions uniform. One of the biggest problems that all this tools have to manage is the heterogeneity of the various providers and of the data obtained. It is, indeed, a goal of these systems the creation of an abstraction level gathering all the common features of the numerous providers.

Hyperic CloudStatus

Hyperic CloudStatus [24] uses Hyperic HQ [25] and providers' specific tools to monitor applications deployed in a multi-cloud context. It provides a view of the health and performance of the most popular cloud services on the web with the goal of identifying the cause when the performance of their cloud-hosted applications changes.

It aggregates multiple metrics from sources inside and outside the cloud to monitor cloud availability and health status. Then it calculates the aggregated data to determine overall availability and normalized metrics across the cloud. The multiple metrics assure to users a relevant overall perspective of cloud performance. For each service, CloudStatus' results reflect general service levels, and serve as an indicator of whether further investigation of application behaviour or cloud performance is required. It is not a low level monitoring components but a powerful dashboard to analyse aggregated data and high-level information.

However, it does not offer any tool to manage the phase of machines creation and neither the machines set up. Another difference from our project is that there is no possibility to undertake actions automatically when a particular situation is verified.

JCatascopia

JCatascopia [6] is another tool with similar functionalities. It is not limited to operate on specific cloud providers and can be utilized to monitor federated cloud

environments where applications are deployed on VMs residing on multiple clouds. It can retrieve heterogeneous information both at machine level (*e.g.,* CPU and Disk usage) and at applications level (*e.g.,* throughput, latency, and availability). It also offers a rule mechanism allowing the developers to aggregate and activate new metrics. Another aspect on which the documentation focuses the attention is the adaptive filtering, performed with the aim of reducing the network and storage overhead by not transmitting values of a metric with very small variance with respect to the previously values.

Another interesting feature of JCatascopia is the possibility to adapt, in a simple way, the monitoring activity after machines migration. Each message contains the IP address of the monitored resource, so at each change the Server is notified. The limitation of this approach is that a request (*e.g.,* metric pull request) issued from a Monitoring Server to an Agent will fail if the IP address is not yet updated. The error space can be shortened if the service issuing the migration informs the Monitoring Agent of this by triggering the right API call. Offering APIs in the Monitoring Server to update the information is a good idea to deal with this problem but, in this solution, they do not offer a deployer that manages the system so the call has to be performed by the user.

Brooklyn

Brooklyn is a framework for modelling, monitoring, and managing applications through autonomic blueprints [5]. A Brooklyn's blueprint defines an application, using a declarative syntax. A blueprint might comprise a single process or encompass combinations of processes across multiple machines and services. These blueprints are modular: blueprints for one process or pattern can be incorporated in other blueprints. Moreover, the blueprints can be treated as source code: tested, tracked, versioned, and hardened as an integral part of the develop process. From a blueprint is possible to deploy to one of many supported clouds or even to multiple different clouds, or to private infrastructure (bring-your-own-node), or to other platforms. Brooklyn will dynamically configure and connect all the different components of an application.

After the deployment it allows to use sensors, effectors and policies provided off-the-shelf (or implement new ones) in order to monitor and manage the running system. Management of monitoring agents (components that are deployed on the monitored machines) is automatic: agents subscribe at the Brooklyn server when

started, while the server check their running status through a heartbeat. According to the documentation, they are planning to offer the possibility to write custom policies to adapt the deployment at run-time but, at the moment, no support is given for adapting Brooklyn monitoring platform based on performance issues.

JClouds

In order to overcome the problem about the heterogeneity of vendors, there are libraries developed with the aim to integrate all the different cloud providers' interfaces. Libraries such JClouds [26] try to integrate all the providers giving an abstraction on the top of them. JCloud is provided by Apache and is an open source library that helps users get started in the cloud. The idea is that the application is described using the concepts and the language offered by JClods and then this code is automatically translated according to the target provider. JClouds, at the moment we are writing, supports 30 cloud providers and cloud software stacks including Amazon, Azure, GoGrid, Ninefold, OpenStack, Rackspace, and vCloud. It offers also the possibility to break the abstraction level and directly use provider API if it is required by the user.

Seaclouds

Seaclouds [27] is a platform built using Brooklyn and JClouds. It supports the application life-cycle management with the ability to dynamically deploy, migrate, replicate, and distribute the modules that compose cloud-based applications among multiple PaaS offering. At the same time, it is capable of checking possible QoS violations and dynamic changes in the offer of the providers and the current demand. A reconfiguration process capable of preserving the soundness of the orchestration, by performing life-cycle management actions when required, permits to adapt coherently the deployment when needed. Moreover, it offers a range of standardized and unified metrics of different types (*i.e.,* low level, container level, and application level) based on disparate underlying cloud providers that will allow the runtime monitoring of deployed services to assure the end-to-end QoS of the complex application. SeaClouds uses Brooklyn to manage the application life-cycle and to monitor it and, on the other side uses JClouds to interact with each cloud provider. However, the project seems to be in an early stage of the development process and no software is offered at the moment.

The problem with all these tools is that no one offers a unique platform with the possibility of managing and monitoring a multi-cloud application and to auto-adapt the monitoring activity at run-time. The tools offered by providers are powerful and complete but are not usable in a multi-cloud context. The other tools lack a monitoring component or they do not provide support for adaptation. Seaclouds is the only software that offers the functionalities that may fulfil all the requirements, but is in an early stage of development. The MODAClouds project proposes to build a single system capable of deploying and monitoring a multi-cloud application and managing the complete life-cycle of the application. Due to its distinctiveness, our contribution is necessary in order to adapt the monitoring activity in any situation of the running system.

Chapter 3

Research problem

"The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency.
The second is that automation applied to an inefficient operation will magnify the inefficiency."

Bill Gates

In this chapter, we describe the research problem and the analysis performed in order to approach the solution. To start the work we had to build an architecture enabling the auto-adaptation and co-evolution of the monitoring activity according to the running system. To obtain this result, we inspired ourselves with a well-known paradigm taken from the literature, therefore very reliable. Once analysed and described the theoretical guidelines underlying the paradigm, we will present in detail each component involved in the system, and finally, with the help of a motivating example used through the entire document, we will show concretely the problem faced and solved.

3.1 The Sense-Plan-Act Paradigm

The Sense-Plan-Act paradigm (see Figure 3.1) is adopted especially in the robotics field, but can be perfectly adapted in the context of the software engineering. It is constituted by three main activities: the *sense*, the *plan*, and the *act*. In robotics, as soon as sensors retrieve information about the environment, the data collected are used by the robot's controller to react to the perceived world computing the

best strategy. The controller is more precisely a set of components with the task to elaborate the action plans of the robot and they are usually structured in a hierarchy. The upper layers in the hierarchy use the world model to make long-term plan at strategic level. The plan is then translated into a sequence of actions and passed to lower levels for the actual short-term execution. Lower layers would then further decompose the commands into more actionable tasks, which are ultimately passed to the actuators at the lowest level. This process is continuous and the sensing layer constantly keeps the internal representation of the world updated. Furthermore, to better accommodate dynamic changes to the world environment, lower planning layers may suggest changes to plans based on recent changes to the world model.

Using this paradigm makes possible to have a strong independence between the three activities (*i.e.,* sense, plan, and act) making the task easier to complete. Moreover, since these operations are executed in a continuous control-loop, the system has always the complete knowledge about the status of the environment.

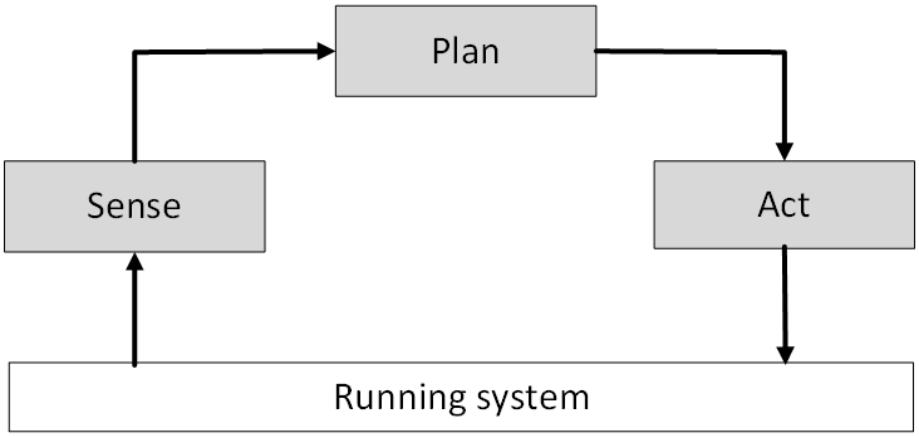

Figure 3.1: Sense-Plan-Act paradigm

The MODAClouds project is a complex system for managing multi-clouds application. As already presented in Chapter 1, the MODAClouds project is composed principally by three components: the Monitoring Platform, the Reasoning Engine, and the Models@Runtime engine. The three components can be perfectly identified as the three actors of the Sense-Plan-Act paradigm. As their names suggest, the Monitoring Platform, in the role of the *sense* activity, monitors the resources retrieving the information from the running system and exposes them

to the Reasoning Engine. The Reasoning Engine, in the role of the *plan* activity, reasons about the status of the running system and decides which changes should be done to adapt the system to the desired behaviour. Finally, the Models@Runtime engine, in the role of the *act* activity, acts directly on the running system, dividing the decisions in subtasks and executing them to reach the target model designed by the *plan* activity.

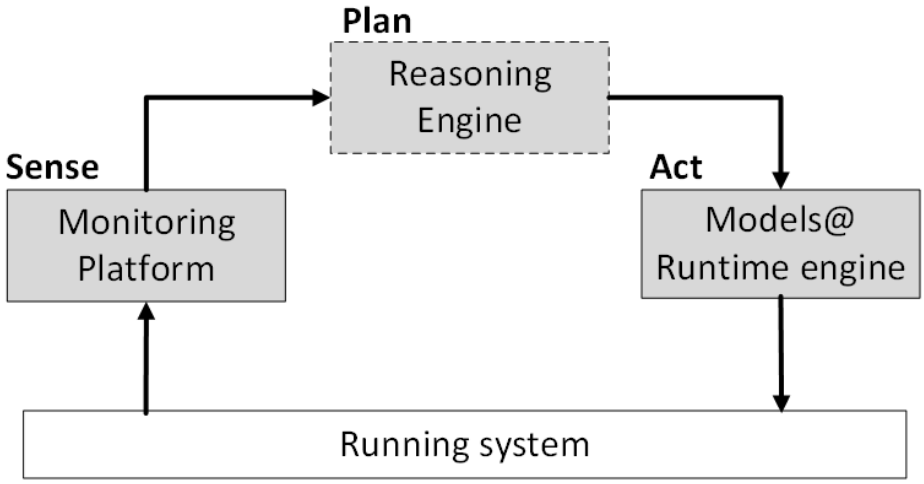

Figure 3.2: Sense-Plan-Act paradigm in the MODAClouds context

The configuration of the Sense-Plan-Act paradigm, in the context of the MODAClouds project, is illustrated in Figure 3.2. The Reasoning Engine box is dashed because we do not deal with the *plan* activity in our work, so there it can be used whatever reasoning engine able to satisfy the interface requirements of the other components. Using the Sense-Plan-Act paradigm permits to build a system that is model driven, in which the components use a representation of the real world to facilitate the operations.

3.2 System architecture

In this section, we present in details the role of each component that we deal with in the MODAClouds project. We describe the components as they were before our contribution, then every modification to the components to achieve the solution will be largely discussed in Chapter 4.

The *acting* activity is monolithic: it is performed by a single component that takes care of all the phases. It is realised by the CloudML Models@Runtime engine, and consists in enacting the provisioning, deployment, and adaptation of multi-clouds systems. This Models@Runtime engine relies on the models@runtime [19, 20] architectural pattern for dynamically adaptive systems. In particular, the Models@Runtime engine provides an abstract representation of the underlying running system.

The *sensing* activity is performed by multiple components acting together in order to monitor the system. It is realised by the MODAClouds Monitoring Platform [28, 29], which is responsible for collecting metrics about the running system through some specific data collectors. The data retrieval process requires the definition of monitoring rules, installed in the Monitoring Platform, describing how incoming data have to be processed, which conditions have to be verified, and which output should be produced. In addition, to support monitoring activity and data interpretation, it requires an ontological representation of the deployment status.

3.2.1 The CloudML Models@Runtime engine

As just introduced, the Models@Runtime engine provides a deployment model related to the running system.

The deployment models are specified in a cloud provider-agnostic way (fundamental to respect the requirement R_1) with CloudML, a language for modelling, provisioning and deploying multi-cloud systems. In particular, CloudML allows expressing the following concepts:

- *Cloud*: Represents a collection of virtual machines offered by a particular cloud provider.
- *Virtual machine*: Represents a reusable type of virtual machine.

- *Application component*: Represents a reusable type of application component to be deployed on a virtual machine, or an external service such as a PaaS solution.
- *Port*: Represents a required or provided interface to a feature of an application component.
- *Relationship*: Represents a communication between *ports* of two application components, or the containment of an application component by another.

Using this language the user can specify a multi-clouds application in the form of types and relationship between them. It is also possible for the user to specify a list of commands that have to be executed on the machine after the provisioning in order to start and configure the deployed applications. Using CloudML is never requested to the user to contact directly the provider asking for a machine and then configuring it. The whole process is mediated by the specification of a model containing all the relevant information and enacting these requests using the deployer engine.

To explain better how Models@Runtime engine works, Figure 3.3 depicts the architecture adopted by CloudML for applying the models@runtime pattern to manage dynamic adaptation.

An external entity, that can be a reasoning engine or the user, reads the current deployment model (step 1), which describes the actual running system, and produces a target deployment model (step 2). The target deployment model represents the new desired configuration of the running system expressed in the form of a complete model and not only as list of added and removed components. Then, the engine calculates the difference between the current model and the target one (step 3). The output of this comparison is a list of actions representing the required changes to transform the current model into the target one. Then, the adapter enacts the adaptation modifying only the parts of the system necessary to account for the difference so that the target model becomes the current model (step 4).

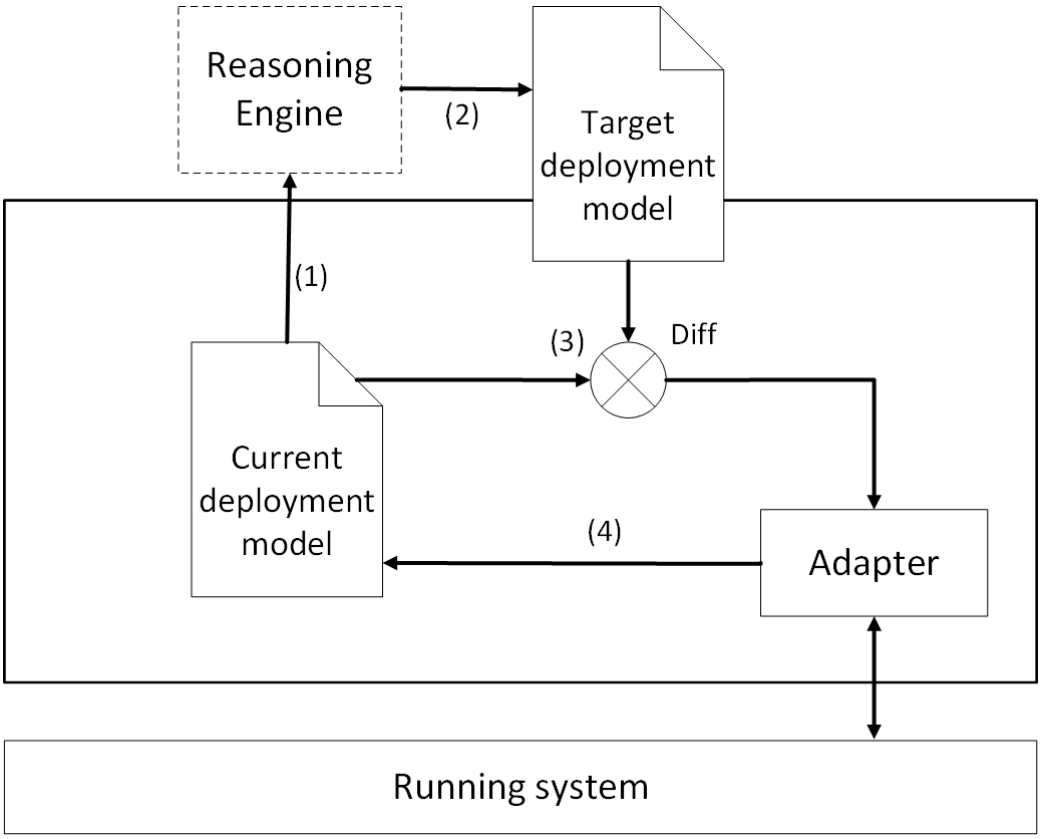

Figure 3.3: CloudML Models@Runtime engine

The strong relationship between the CloudML Models@Runtime engine and the running system ensures that any modification of the model is propagated, on-demand, onto the running system. In this way, the user or any external entity that wants to change the status of the system can provide a new model representing the desired configuration and the engine will act on the running system to perform the requested changes.

3.2.2 The MODACloudsMonitoring Platform

We are now going to present each element involved in the *sensing* activity. Figure 3.4 depicts the Monitoring Platform with all the elements involved in monitoring the resources. The Monitoring Platform is composed by several components that we are going to illustrate in this section: the Data collectors collect data from the running system, the Knowledge Base stores the model of the running system, the Data Analyzer aggregates the data collected, the Monitoring Platform configures all the other components, and finally the Metric Observer receives the data flow from the Data Analyzer.

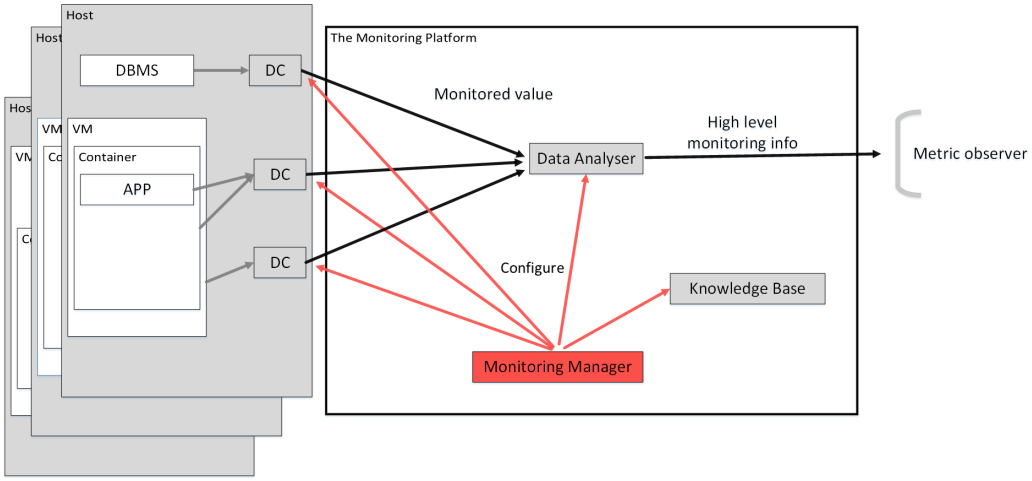

Figure 3.4: Monitoring Platform architecture

The Data Collectors

The Data Collectors (DCs) are responsible for collecting data from various sources (see Chapter 5 for implementation details). The DCs can be split into two main categories: application level DCs and infrastructural level DCs.

The Data Collectors at application level are focused on collecting data related to the application execution. The application level metrics are values related to the performances of the running applications and their supporting platforms (*e.g.,* arrival rate, throughput, error rate, response time). The Data Collector at application level is a library that must be included as a dependency into the application to monitor. In order to retrieve values about application level metrics, the methods of

the application must be annotated directly in the code with dedicated notations that enable the collection of values.

The Data Collectors at infrastructural level are low-level collectors and are focused on collecting information representing the status of the environment in which the applications run. The infrastructural level metrics are related to the state of the virtual machine (*e.g.,* CPU usage, memory used, disk space used, read/write operations on disk) and are retrieved directly from the machine and from the IaaS (Infrastructure as a Service) provider. In order to work properly, the infrastructural Data Collector must be directly launched on the machine to monitor. It uses libraries and components that permit to retrieve information independently from which cloud provider is hosting the machine.

Both types of Data Collectors retrieve their configuration from the Knowledge Base to start collecting data automatically. They look for a configuration compatible with the name of the resource that they are monitoring, therefore it is fundamental that the Knowledge Base contains the exact names of the resources in the running system, otherwise they cannot start monitoring.

The Knowledge Base

The model representation of the running system and the DCs' configuration to start the monitoring activity are stored in The Knowledge Base (KB). Apache Fuseki [30] was adopted as RDF triple store for the Knowledge Base. It allows access through REST APIs to extract and insert data. The ontology stored in the KB describes components and their relationships, and it is used to interpret monitoring data. Compared to the models used within the Models@Runtime engine (specialized in the deployment domain), the KB uses an enriched ontological model (specialized in the monitoring domain) which is also cloud provider-agnostic (R_1).

As said, this model is retrieved in pull mode by Data Collectors to reconfigure themselves accordingly; if the model contained in the KB is not synchronised with the running system the DCs are not able to start monitoring the resources.

The Monitoring Manager

The Monitoring Manager is the main coordinator of the monitoring activity; it manages and configures all the monitoring components. It can be contacted, using the exposed interfaces, to modify the monitoring activity, and then it acts on the other components updating them with the new specifications of the monitoring. The Monitoring Manager offers HTTP REST APIs that can be used to perform the following actions:

- Retrieve the list of installed monitoring rules.
- Install a monitoring rule.
- Delete a monitoring rule.
- Retrieve the list of metrics available to be observed.
- Retrieve the observers attached to a specific metric.
- Attach an observer to a specific metric.
- Detach an observer from a specific metric.

It is important to notice that using the features of the Monitoring Manager it is possible to perform parameter adaptations. Indeed, the monitoring rules can be modified at run-time according to the specifications of the user, so the metric monitored, or the way to collect it, can be varied.

Monitoring rules

A Monitoring Rule is the recipe that describes what the platform should monitor, how data should be aggregated, what conditions should be verified, and what actions should be performed whenever the condition is verified.

More in detail a rule contains the following fields:

- *timeStep:* interval between two following evaluations of the rule
- *timeWndow:* time range in which data have to be considered for aggregation
- *monitoredTargets:* the list of resources to be monitored
- *collectedMetrics:* the list of metrics to be collected on the target resources

- *metricAggregation:* any aggregation the platform should do on data
- *condition:* a boolean condition predicating on the aggregated value
- *actions:* a list of actions to be executed whenever the condition is verified

In the description of the motivating example (see Section 3.4), a complete monitoring rule will be shown and commented.

The Data Analyser

The Data Analyser (DA) is the component in charge of analysing the monitoring data. It adopts C-SPARQL as query language and C-SPARQL Engine [31] as RDF stream processor. The DA executes C-SPARQL queries generated according to the rules installed in the Monitoring Manager (see Section 3.2.2). The installation of a query generated from a monitoring rule is performed by the Monitoring Manager in completely transparency to the user.

The DA processes at real-time the RDF monitoring data coming from the DCs, interprets, filters and aggregates them, basing on the installed queries rules and on the system state stored in the KB. Moreover, it detects violations based on constraints described in the monitoring rules and serializes data to be delivered to observers (see Section 3.2.2). The DA is currently able to perform simple aggregations such as the average, count, and extraction of minimum, and maximum.

The Metric Observers

A Metric Observer receives the data flow coming from the DA. It can be registered to any available metric and must expose an HTTP REST API to receive the data flow. Any external third-party entity (*e.g.,* reasoning engine) interested in the data can declare its own Metric Observer and register it to the Monitoring Manager in order to receive high level information to perform some operations on them.

3.3 Adaptation

Dynamic adaptation modifies the behaviour of an application either by changing a parameter in the application components (parameter adaptation) or by reconfiguring the application to replace, update, or modify the set of its components (compositional adaptation) [32] at runtime. In the same way, the monitoring platform of MODAClouds can be adapted modifying parameters (*e.g.,* sampling time, sampling probability, enabling or disabling a monitoring rule) or applying structural adaptation (*e.g.,* deploy, remove, or migrate part of the monitoring platform together with the application).

In respect of the parameter adaptation, the platform can be adapted at run-time through the tuning, disabling and enabling of the monitoring rules that are already installed on the Monitoring Platform, and through the dynamic addition/deletion of the monitoring rules.

Each rule contains a set of parameters that can be changed to modify the behaviour of the Data Collectors at run-time.

This adaptation can be performed by setting the sampling time and the sampling probability of each rule, modifying the way in which data are collected. Changing the sampling time affects the frequency of the metric acquisition. This kind of adaptation is useful when the goal is to reduce the impact of the monitoring activity on the hosting machine. Conversely, the sampling time can be increased in order to have more accurate data at the cost of higher computational overhead.

Monitoring rules can be also enabled or disabled, restoring or interrupting the outgoing data flow from the Data Collectors. There is also the possibility to trigger an action when a particular condition is violated, so that the sampling time and sampling probability can be automatically modified, or a monitoring rule can be enabled or disabled.

In order to modify the way in which the data are processed by the Data Analyser, it is also possible to change the rules (*e.g.,* moving from the calculation of the average of a metric to the calculation of the maximum or minimum value registered by the Data Collector).

The last possible way to perform parameter adaptation is to modify the set of outgoing metrics, according to the components and to the third-parties entities needs, by installing and deleting monitoring rules.

The communication with the Monitoring Platform is performed using the interfaces provided by the Monitoring Manager.

A rule can be installed by sending a Post request to the Monitoring Manager, and can be deleted with a Delete request.

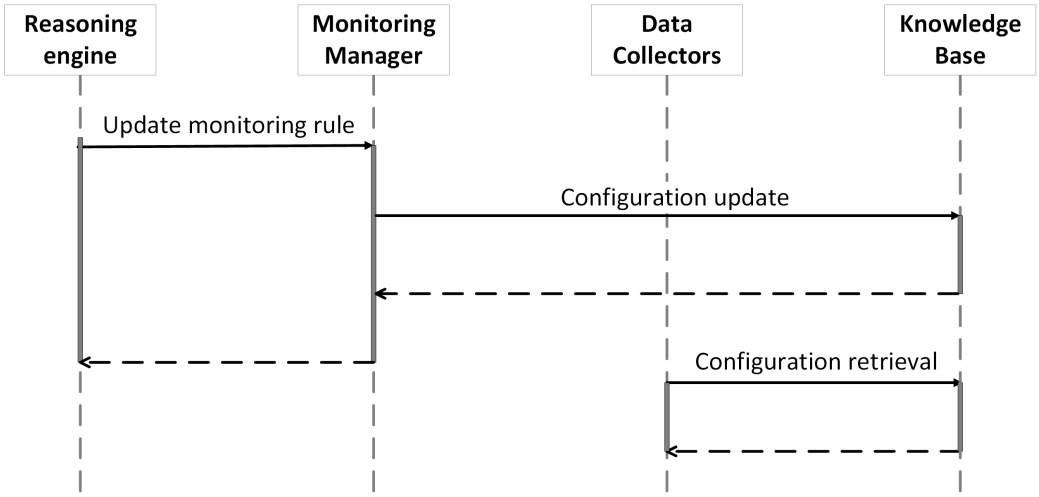

Figure 3.5: Sequence diagram for setting the sampling time of a monitoring rule

The typical interactions between the various components of our platform when a rule is modified in the Monitoring Platform are shown in Figure 3.5. It requests to the Monitoring Manager, through the API exposed, to modify the sampling time of the rule. The Monitoring Manager automatically updates the monitoring rule with the preferred sampling time storing the configuration of the Data Collectors in the Knowledge Base. Each Data Collector will pull its own configuration, at every predefined interval time, requesting it to the Knowledge Base.

What MODAClouds was lacking, before our contribution, is the capability of performing structural adaptation, in other words the system could not react to modification in the structure of the running system and could not monitor new resources. The motivating example in the next section will describe better the situation.

3.4 Motivating example

In this section, we present a motivating example that reflects a real case scenario in the multi-cloud environment. We proceed through various steps that show the possible issues that may arise using the architecture presented in this chapter.

The example consists in the deployment of MIC (*i.e.,* "Meeting In the Cloud"), first on a single cloud, and then on multiple clouds. MIC is a Java web application developed by the Politecnico di Milano that permits to users to find and meet other people with the same passions. It is a kind of social network; each user has to register in order to exploit the functionalities offered by the web application, once registered, the user has to write his own hobbies and express his interest, in a scale from one to five, on various topics.

The architecture of MIC is simple; it is composed by three main components: a frontend application exposing the methods accessible and the graphic interface, a backend that computes the similarity between the users, and a database in which the users' data are stored. The two applications, frontend and backend, are Java EE applications and need a Java application server as container to be executed. In the context of this example we use GlassFish [33] provided by Oracle. Another requirement is that, on the machine that hosts the frontend application, an instance of Memcached has to be running. Memcached [34] is a memory object caching system used to improve the performances. Figure 3.6 shows the deployment model in a graphical way. The black arrows visualize the dependency between the components and the nested configuration points out that MIC runs in the GlassFish container.

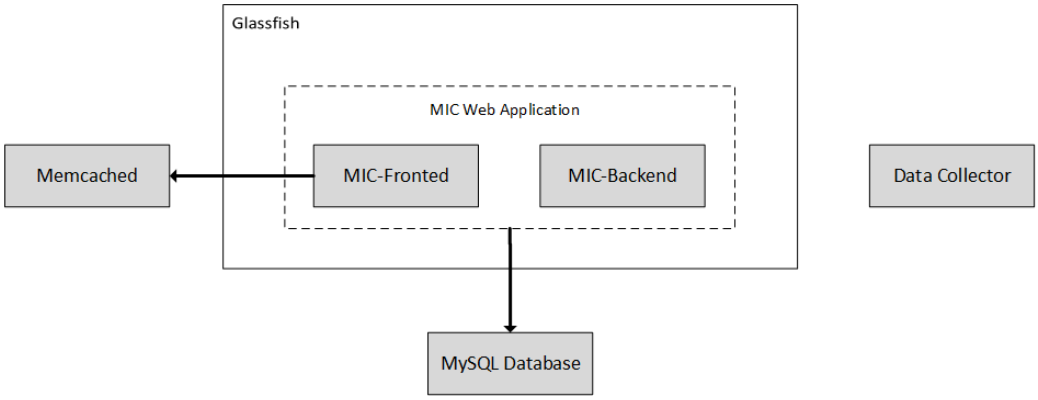

Figure 3.6: MIC architecture

We chose to use MIC for our tests because it is an application with the right level of complexity. It is not composed by a single component but has a small number of dependencies that can be easily managed. Nevertheless, MIC is complex enough to ensure that the results are actually useful; its architecture with two Java applications, a Database, and some extra requirements are a good approximation of the majority of available applications.

An important point that we want to stress is that the results obtained are extendible to applications different from MIC, because they are independent from the particular application and from the operations performed in the example.

As depicted in Figure 3.7, initially, the deployment is done on a single cloud provider: Flexiant. The deployment is performed by the Models@Runtime engine that sets up the system according to the model specified by the user. The engine takes care of provisioning the machine and then installs all the needed software on it. The configuration of both the applications and the machine is done according to the commands declared by the user during design phase. In order to start monitoring the running system, a data collector must be installed and started on the VM to retrieve information about the underlying system. Once that the deployment is completed and the data collector is installed, we have the situation described in Figure 3.7.

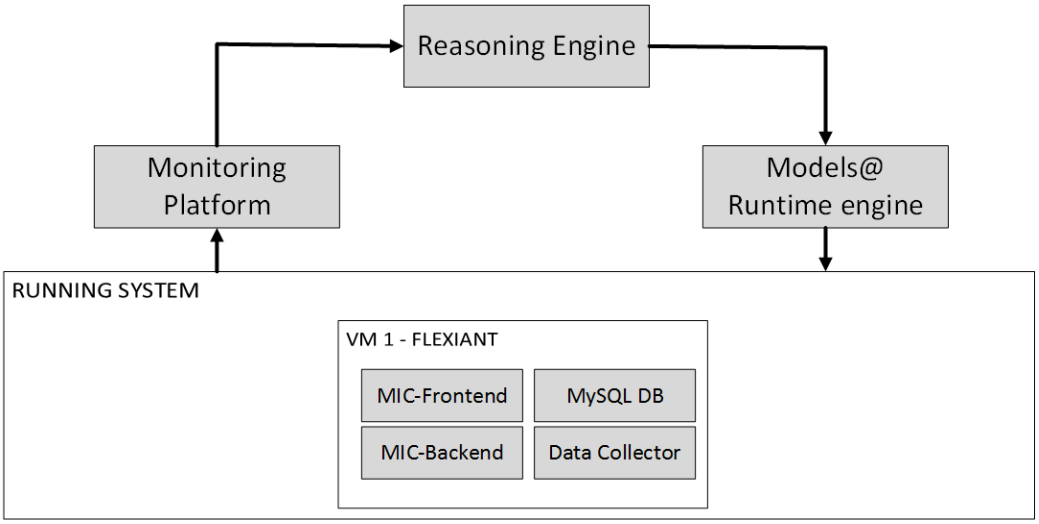

Figure 3.7: Initial configuration of the scenario

The next step is to start monitoring the resources of the running system at infrastructural and application level. In order to do that, we decide to install two

monitoring rules, one to monitor the CPU load on the resource, and the other one to monitor the response time of a specific method of the application.

Listing 3.1 presents a rule to monitor the CPU utilization of the VMs to better clarify the way in which the Monitoring Platform works:

Listing 3.1: Monitoring rule for the CPU usage

```
<monitoringRules
xmlns="http://www.modaclouds.eu/xsd/1.0/monitoring_rules_schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.modaclouds.eu/xsd/1.0/monitoring_rules_schema">
        <monitoringRule
       timeWindow="15"
       timeStep="15"
       label="CPU rule"
       id="cpuRule">
                <monitoredTargets>
                        <monitoredTarget class="VM" type="MIC-Node"/>
                </monitoredTargets>
                <collectedMetric metricName="CPUUtilization">
                        <parameter name="samplingTime">5
                </parameter>
                </collectedMetric>
                <metricAggregation
           aggregateFunction="Average"
           groupingClass="CloudProvider"/>
                <condition>METRIC &gt;= 0.2
           </condition>
                <actions>
                        <action name="OutputMetric">
                                <parameter name="name">CPUUtilizationViolation
                   </parameter>
                    <parameter name="value">METRIC</parameter>
                                <parameter name="resourceId">ID</parameter>
                        </action>
                </actions>
        </monitoringRule>
</monitoringRules>
```

The rule specifies that the CPU usage has to be collected every 5 seconds and that a result has to be produced on the output stream only if the average usage during the last 15 seconds is *greater than* 20%. It affects all the virtual machines (*i.e.,* class="VM") whose type is *MIC-Node*. The rule is evaluated every 15 seconds and the result is a new monitoring datum with *CpuUtilizationViolation* as metric name, the average CPU utilisation as value, and the name of the cloud provider where the failure occurred as target.

As the listing shows, the name of the metric of interest and some parameters

(*e.g.,* the sampling time) are specified when defining the *collectedMetric*. The *aggregationFunction* field inserted in the rule specifies which mathematical operation has to be computed during the data processing (*i.e.,* average, count, minimum, maximum, and percentile). Each time the *condition* is violated, the action declared in the *action name* is performed (*i.e., OutputMetric*, *EnableMonitoringRule*, *DisableMonitoringRule*, *SetSamplingProbability*, and *SetSamplingTime*).

An example of monitoring rule for the application level monitoring is presented in Listing 3.2. The monitoring rule's *id* is *respTimeRule* because it evaluates, as specified in the *metricName*, the response time of various methods offered by the application. The monitored targets are the methods named *answerQuestion*, *saveAnswers*, and *register*, and the output metric contains the average response time of each one.

Listing 3.2: Monitoring rule for the response time

```
<monitoringRules
xmlns="http://www.modaclouds.eu/xsd/1.0/monitoring_rules_schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.modaclouds.eu/xsd/1.0/monitoring_rules_schema">
  <monitoringRule
  timeWindow="30"
  timeStep="30"
  id="respTimeRule">
    <monitoredTargets>
      <monitoredTarget class="Method" type="answerQuestions" />
      <monitoredTarget class="Method" type="saveAnswers" />
      <monitoredTarget class="Method" type="register" />
    </monitoredTargets>
    <collectedMetric metricName="ResponseTimes">
      <parameter name="samplingProbability">1</parameter>
    </collectedMetric>
    <metricAggregation aggregateFunction="Average"
    groupingClass="Method" />
    <actions>
      <action name="OutputMetric">
        <parameter name="metric">AverageResponseTime</parameter>
        <parameter name="value">METRIC</parameter>
        <parameter name="resourceId">ID</parameter>
      </action>
    </actions>
  </monitoringRule>
</monitoringRules>
```

A first problem arises once that the rules are installed. The Monitoring Platform fetches from the Knowledge Base the information to configure the Data Collectors and to start the monitoring, but if the model stored in the Knowledge Base is not

consistent with the reality, the Monitoring Platform does not know which resources and which types are deployed in the running system, therefore, the DCs cannot start the monitoring activity.

Let us assume that, at certain point, the CPU load value on the machine becomes too high for the requirements of the system and the Reasoning Engine decides to migrate the frontend application to another VM to decrease the load, and for costs reason, the migration will be performed on a different cloud provider: Amazon EC2.

The Models@Runtime engine asks for the creation of a new virtual machine and installs the components, enacting on the running system the orders of the reasoning engine. The new configuration is that the VM1 contains the backend and the databases, and the frontend is moved to another machine. If nobody advises the Monitoring Platform that the migration has been performed, the data collectors will not be able to re-configure themselves in order to monitor the new resources. Moreover, not only the data collectors are not configured properly, but the Monitoring Platform believes that the deployment is still composed just by a single machine. This situation is depicted in Figure 3.8 in which the question marks indicate that these components are not correctly stored in the Knowledge Base. The problem with the platform resides in the fact that the Models@Runtime engine cannot communicate what it changed in the running system.

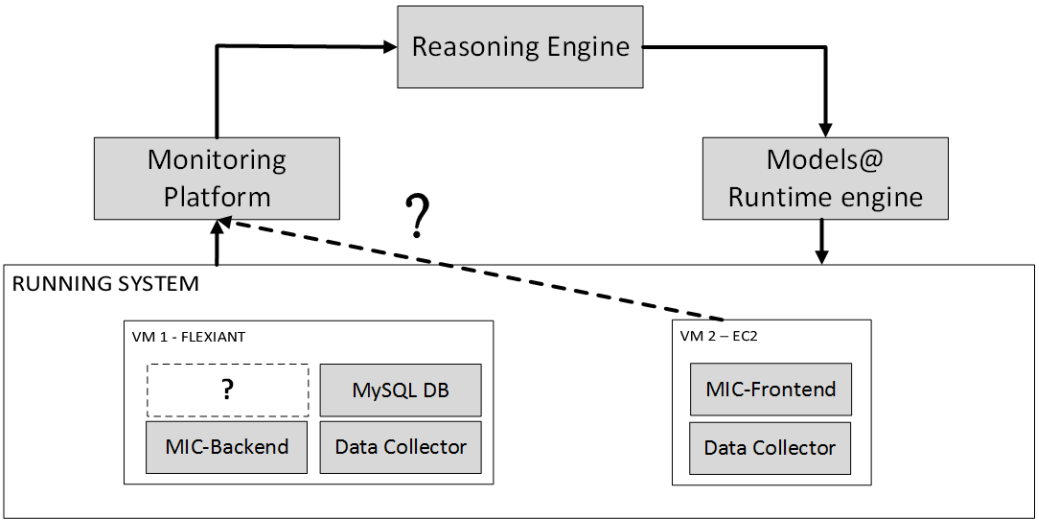

Figure 3.8: Final configuration of the scenario

Let us suppose, as last step of the motivating example that some external event causes a problem with the VM2 and that it falls in a *STOPPED* status, declared by the provider. The Models@Runtime engine is not capable of recognising the new status of the machine and cannot update its own model to set its new status. The situation is that the model representing the system does not reflect the real situation of the deployment, because, from the model point of view, the VM2 is still running but, in reality, it is *STOPPED*. This does not permit to the Reasoning Engine, which observes the Models@Runtime engine's model, to react to this situation.

3.5 Requirements

The above scenario helps us in defining better the requirements already presented, analysing them in the MODAClouds context:

- **Cloud provider-independence (R_1)**

  Our platform should support a cloud provider-agnostic specification of the monitoring, provisioning, and deployment of multi-clouds applications. This simplifies the design and the adaptation of both the multi-clouds application and the monitoring platform, and prevents vendor lock-in. Both the monitoring activity (*i.e.,* the Monitoring Platform) and the deployment activity (*i.e.,* CloudML) must work with any cloud provider without the need to change anything in the model or the platform.

- **Model abstraction and coherence (R_2)**

  Our platform should provide an up-to-date, abstract representation of both the running system and the monitoring processes. This will facilitate the reasoning, simulation, and validation of adaptation actions before their actual enactments. Our platform should update the representation of the system after any change in the model. The changes can be performed by the system itself (*e.g.,* a new machine is added) or may happen for external causes (*e.g.,* power loss). Only by having a model always consistent with the running system we can ensure that the platform evolves in a right way.

- **Co-evolution of the monitoring model with the application (R_3)**

  Our platform should maintain the monitoring model consistent with the actual deployment of the running system. This will avoid to the user the need of reconfiguring manually the Monitoring Platform after changes in the deployment. The model stored in the Knowledge Base must be consistent with the model of the Models@Runtime engine. Every time a modification in the deployment model occurs, the Models@Runtime engine must communicate to the Monitoring Platform which resources are affected by the changes.

- **Adaptation of the monitoring platform at run-time (R_4)**

  Our platform should provide support for dynamic adaptation of the monitoring platform. This will permit to modify the monitoring process at run-time. Storing correctly the updates that the Monitoring Platform receives, permits to the Monitoring Platform to configure properly all the data collectors present in the deployment model, and to start automatically the monitoring of the new resources according to the monitoring rules defined.

In the next chapters (Chapter 4 and Chapter 5), we present our approach and how it addresses these requirements.

Chapter 4

Solution

"There are three principal means of acquiring knowledge: observation of nature, reflection, and experimentation. Observation collects facts.
Reflection combines them.
Experimentation verifies the result of that combination."

Denis Diderot

In Chapter 3, we presented the MODAClouds project before our contribution and, with the help of a motivating example, we identified some problems that may arise if certain situations happen. In this chapter, we explain the architecture proposed by us to solve these issues and our design choices.

4.1 Model synchronisation

Analysing the requirements described in Section 3.5 we started from the classical Sense-Plan-Act paradigm and extended it, in order to permit the dynamic adaptation of the monitoring activity. What is clear from the analyses performed in Chapter 3 is that there is a lack of a communication channel between the Monitoring Platform and the Models@Runtime engine in the sense-plan-act paradigm adopted it in the MODAClouds context. To solve the problems, it should be possible to advise the Monitoring Platform about any modification of the model performed by the Models@Runtime engine, since the first deployment to every modification at runtime. For this reason, we created a bidirectional communication channel

coupling the two components; this permits to synchronize and enrich the representations of the system of the Monitoring Platform and the Models@Runtime. In Figure 4.1, the grey lines show where our contribution takes effect in the extension of the paradigm.

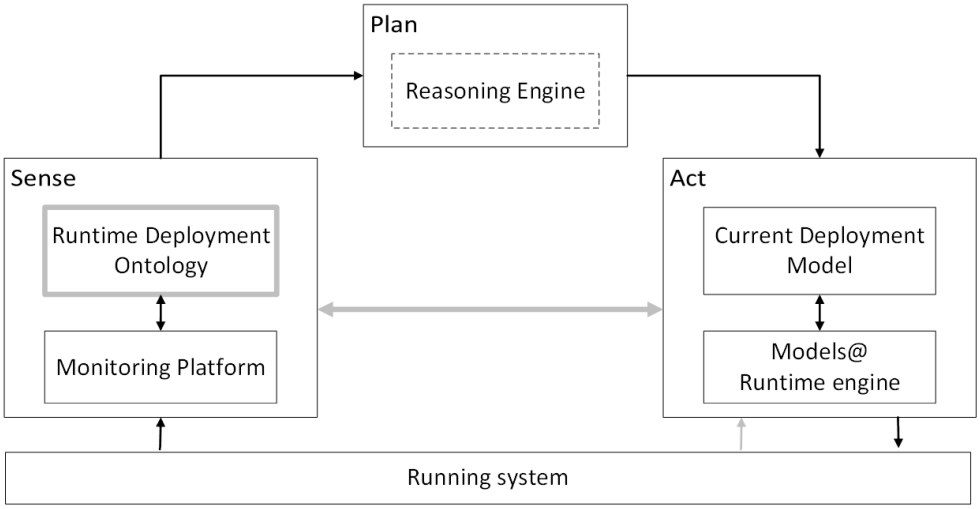

Figure 4.1: Extension of the Sense-Plan-Act paradigm

As we have already described, the Models@Runtime engine and the Monitoring Platform are model based and, since the domain of the two models are different, each of them contains a slightly different representation of the system to ensure the correct level of abstraction. This produces a clear separation between the *sensing*, and the *acting* activities, making the system more robust to failures, as each of them can be executed independently using the information coming from its own model. However, this separation may create inconsistencies between the model in the Monitoring Platform and the running system. Creating a communication channel between the two components and synchronizing them, our approach transforms the static model of the Monitoring Platform (depicted in the pale-grey box) into a runtime deployment ontology (*i.e.,* constantly consistent with the status of the running system) thus enabling the automatic adaptation of the sensing activities. In this way, both the models are changing according to the status of the running system and the Monitoring Platform can adapt the monitoring activity.

To ensure the independence and the robustness of the two components, we also added to Models@Runtime the ability to retrieve some information directly from the running system (little grey arrow). More in details, we added a component

that monitors the status of machines and updates the model if something changes. This solves the problem of losing the consistency of the model if the user manually deletes a machine or if it goes off-line for any other reason.

Before starting with the design phase of the APIs, we studied which was the best logic to synchronise the models. CloudML enacts all the changes in the deployment model. For this reason, we decided to implement the communication in the form of push notifications from the Models@Runtime engine to the Monitoring Platform. In this way, we ensure that, as soon as something is changed in the deployment model, the information is sent with a notification, guaranteeing the minimum network traffic overhead.

The opposite logic, asking at a given frequency to the Models@Runtime engine the changes in the deployment, presents some weak points: if nothing changes in the deployment, a request coming from the Monitoring Platform is performed anyhow and has to be sent on the network and managed by the Models@Runtime engine. In addition, this logic introduces a delay in the system: supposing that the interval between two consecutive requests is x seconds, without considering any other delay, the change is detected from the Monitoring Platform in average $x/2$ seconds after that is enacted on the system.

To receive the messages about the changes in the running system, the Monitoring Platform needs dedicated interfaces. We designed multiple interfaces, due to the different types of modification that can happen during the life-cycle of an application.

We implemented the following APIs:

- **Put a deployment model**, used to upload a deployment model from scratch, usually invoked at the first deployment of an application or to replace the whole model.
- **Post resources to the Monitoring Platform's model**, used to update the model sending any modification that involves the creation or the update of resources to monitor at runtime.
- **Delete a resource from the Monitoring Platform's model**, used to send any modification that involves the removal of resources that have no longer to be monitored.
- **Get a resource from the Monitoring Platform's model**, used to retrieve information about a specific resource.

Every time one of these APIs is called, the Monitoring Platform has to handle the information and update the Knowledge Base respecting the internal dependencies.

On the other side, as shown in Figure 4.2, also the Models@Runtime engine needs additional components to ensure the model synchronisation. It needs a component capable of actually managing the network communication and sending the updates on the communication channel. Considering the domain difference between the two models, before sending the updates the model has to be translated into a format compatible with the one of the Monitoring Platform. For this reason, a component should also manage the translation of the model from the Models@Runtime engine syntax to the Monitoring Platform one.

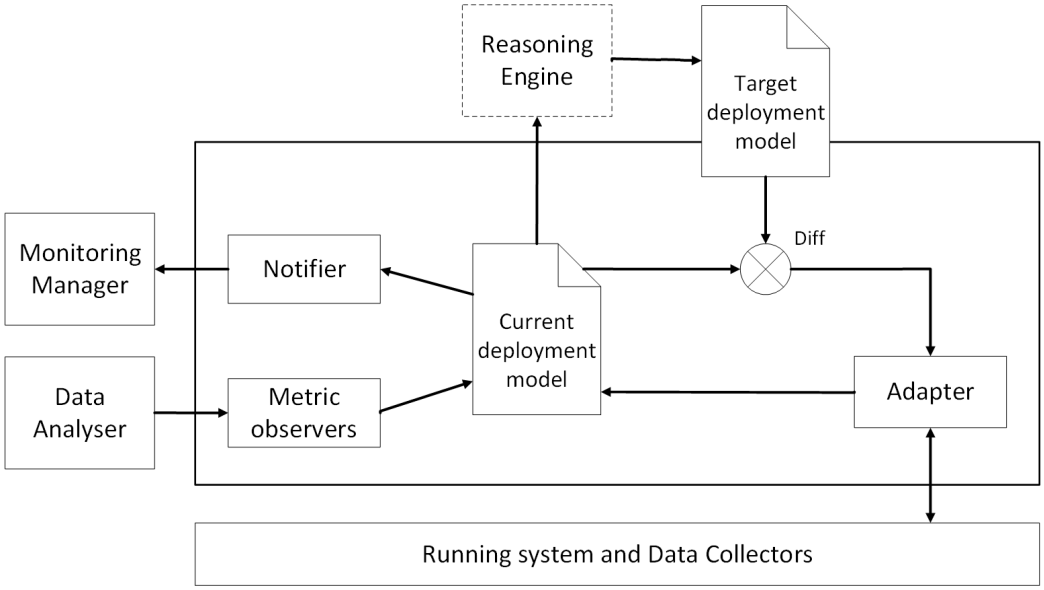

Figure 4.2: Models@Runtime engine architecture

In addition, the Models@Runtime engine needs a component to retrieve the status of the running VMs to have a complete representation of the running system (see requirement (R_2)). The Status Monitor controls the real status of the machine at runtime, updating the Models@Runtime's model in case of changes.

Furthermore, in our architecture, we imported a Metric Observer in the Models@Runtime engine, enabling the possibility to enrich the model with data coming from the monitoring activity. We have not exploited this functionality but in our opinion it might be useful to add to the Models@Runtime engine model also some

high level data of the monitoring activity (*e.g.,* CPU average usage in the last minutes) to simplify the work of external entities that can retrieve the data from a unique model.

The final architecture proposed is illustrated in Figure 4.3. The Models@Runtime engine has its own current deployment model, plus a metric observer that enriches the model with high-level monitored values. Furthermore, the engine can retrieve the status of the VM updating the model if necessary. The current model is offered to the reasoning engine (or a user) that produces a target model as output. The deployer acts on the running system, performing such actions that allow converting the target model into the current one. When the modification of the deployment is concluded, the Models@Runtime engine sends the updates to the Monitoring Platform, which keeps up-to-date the model stored in the Knowledge Base. In this way, the deployed DCs can retrieve their updated configuration from the Monitoring Platform, confident that the model corresponds with the running system.

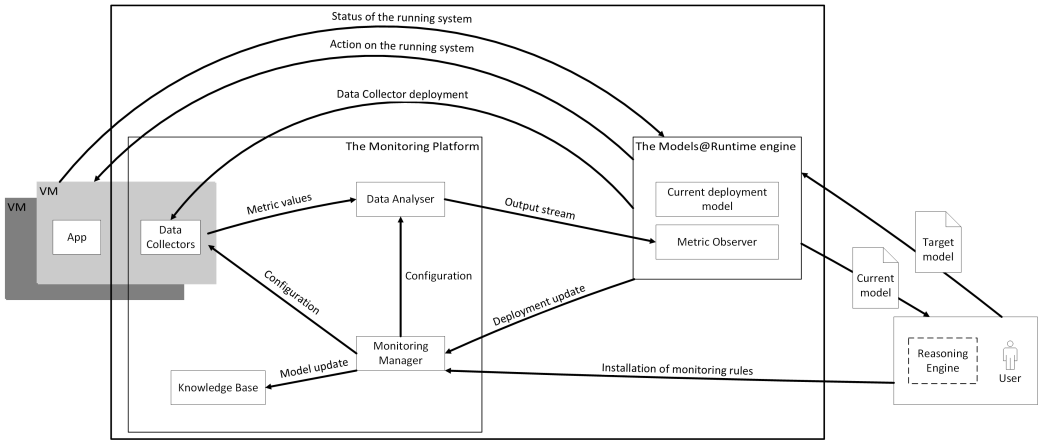

Figure 4.3: Overall architecture

The defined architecture offers the possibility to add any reasoning engine as an external module as long as it is integrated properly through the exposed interfaces. Moreover, the model-based approach adopted provides such external module with well-defined interfaces, and the abstractions offered enable reasoning engines and users to analyse a simplified and domain-specific representations of the running system (R_2).

4.2 Models coherence

Once defined the synchronisation strategy, we start considering what kind of information should be transmitted between the Monitoring Platform and the Models@Runtime engine. The main goal of the work is to keep the model inside the Monitoring Platform always updated and consistent with the status of the running system. As previously written, the domains of the two models are different, so there is the need to translate from one model to the other. We are now going to present the two models, as well as the logic behind the translation we performed.

4.2.1 CloudML model

CloudML implements the type-instance pattern [35], which facilitates the reuse and the abstraction of the resources. This pattern utilizes two flavours of typing, namely ontological and linguistic, respectively [36]. In Figure 4.4, it is represented the UML diagram for the type model.

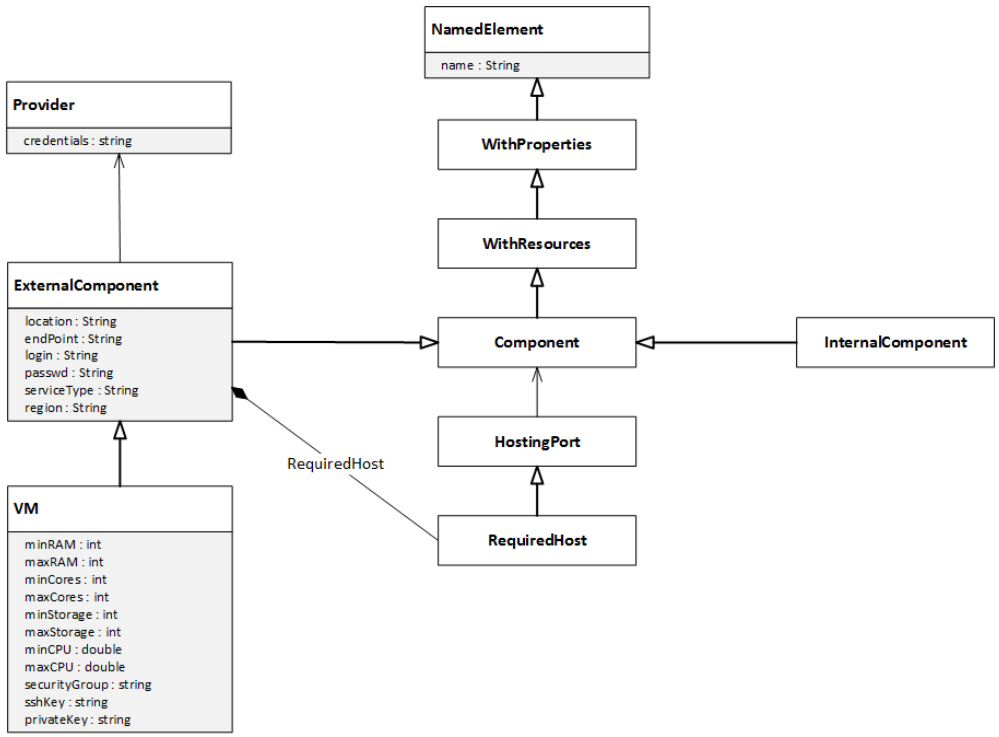

Figure 4.4: Type part of the CloudML model

Every resource extends the NamedElement class, in which it is saved the name of the element. CloudML guarantees that the name is unique for each resource. A Component represents a reusable type of component of a cloud-based application, this concept has been refined with subtypes. A Component can be an ExternalComponent, meaning that it is managed by an external Provider, or an InternalComponent, meaning that it is managed by CloudML. This mechanism enables supporting both IaaS and PaaS solutions through the single abstract concept of component. The property *location* of ExternalComponent represents the geographical location of the data centre hosting it. The properties *credentials* and *serviceType* represent the authentication information needed to access to this specific service and its type (e.g., database, application container). An ExternalComponent can be a VM. The properties *minCores*, *maxCores*, *minRam*, *maxRam*, *minStorage*, and *maxStorage* depict the lower and upper bounds of virtual compute cores, RAM, and storage of the required virtual machine. The property OS represents the operating system to be run by the virtual machine. A HostingPort represents a hosting interface of a component. A HostingPort can be a ProvidedHost, meaning that it provides hosting facilities (*i.e.,* it provides an execution environment) to another component (*e.g.,* a virtual machine running GNU/Linux provides hosting to a JEE container), or a RequiredHost, meaning that an internal component requires hosting from another component.

These types can be instantiated in order to form an assembly of components that specifies a deployment model. Each instance is identified by a unique identifier and refers to a type (see Figure 4.5)

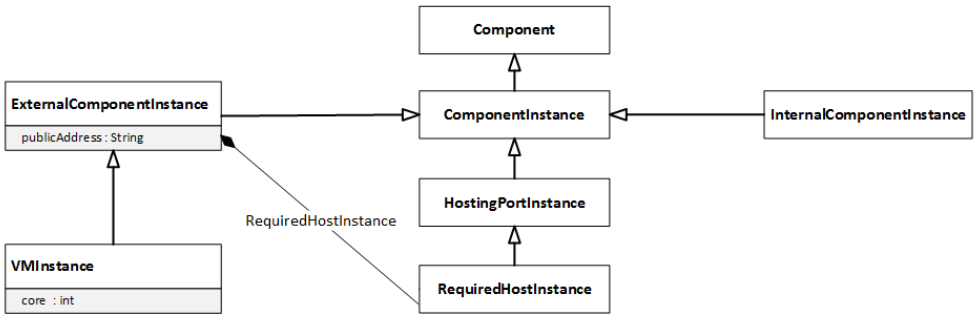

Figure 4.5: Instances of the CloudML model

It is important to note that the number of cores and the public address of an external component are stored in the ExternalComponentInstance, because they cannot be declared at design time, but can be retrieved only at run-time.

4.2.2 Knowledge Base model

The model contained in the Knowledge Base, presented in Figure 4.6, specifies all the needed information to monitor the resources in the deployment model.

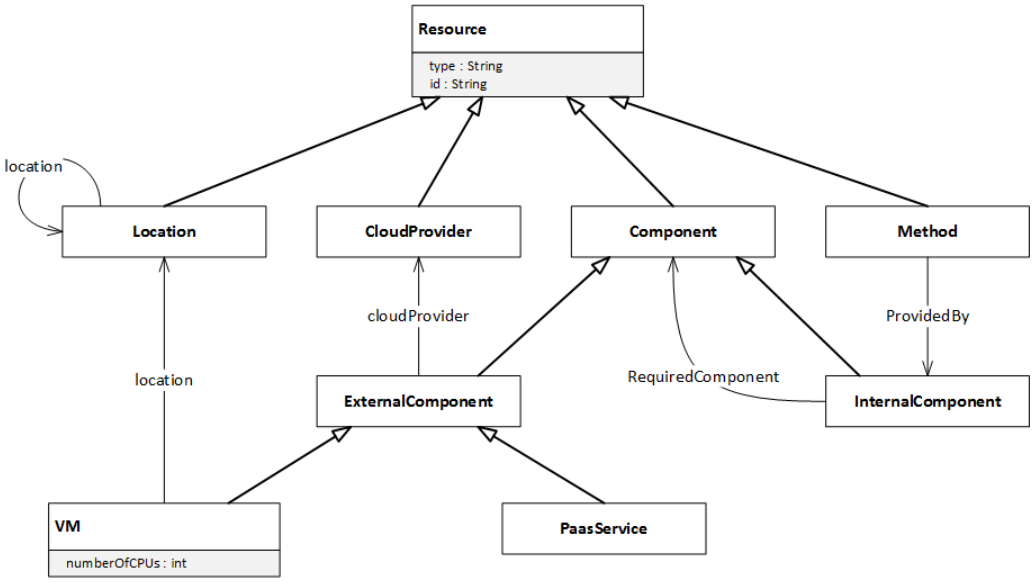

Figure 4.6: Model stored in the Knowledge Base

Every resource extends the Resource class. The Resource class contains two properties: the *id* and the *type*. The *id* identifies in a unique way the resource in the whole model, and the *type* refers to the nature of the resource. A resource can be a Component, a Method, a Provider, or a Location. They can be further split in ExternalComponent and InternalComponent. Respectively, a Data Collector at infrastructure level monitors an ExternalComponent, and a Data Collector at application level monitors an InternalComponent. Each ExternalComponent contains a property *CloudProvider* referring to the hosting provider. The ExternalComponent is an abstract class, so it has to be extended by VM or by PaasService. The VM contains the property *numberOfCPUs* that must be defined at runtime once that the machine has been deployed. Moreover, it has a relation with the *location*, which indicates the geographical area where the VM is located. Any InternalComponent has a property requiredComponents with the IDs of the components needed for the execution (*e.g.,* the host VM) The Method contains the property *providedBy* with the id of the Component that offer it.

4.2.3 Models integration

We can pass now to the comparison and integration of the models. Since the configuration of the running system is stored in the Models@Runtime's engine model we map the information contained in it into the Monitoring Platform's one. The models and the abstraction levels of the models are different because the tasks of the components are completely unrelated. The Monitoring Platform needs the model to monitor the resources, the Models@Runtime engine stores the model to act on it. A clear example of this difference is the information about methods. For the Models@Runtime is irrelevant which methods an application offers, as long as all the dependencies between applications are specified. Instead, this information is required by the Monitoring Platform to monitor the performances of applications.

Since the models contain different information and have different semantics, the only solution is to implement a component that maps the CloudML's model in a format compatible with the Monitoring Platform's one. We could place this translator in the Models@Runtime or in the Monitoring Manager without any significant difference at conceptual level (*i.e.,* we could translate and then send or we could send and then translate). Nevertheless, since this component not only performs model translation, but also filters all the non-relevant information for the monitoring activity, we decided to implement it as a module of the Models@Runtime engine. In this way, we can obtain lower network overhead and a faster communication sending only the significant information.

Each resource in the Knowledge Base has its own attribute *id* and it must be unambiguous. On the other side, also in CloudML each resource has the *name* attribute and it is unique. Thus, we can simply map the name of a CloudML resource into an id for the Monitoring Platform. The *type* attribute can be directly retrieved from the type of instance declared in the CloudML model. Since the pattern followed is the type-instance, each instance has for sure a type from which it has been generated. In the Knowledge Base, the ExternalComponent is an abstract class, so no object can be instantiated as ExternalComponent class. The *publicAddress* attribute of the VM class can be retrieved from the ExternalComponentInstance of the Monitoring Platform model. The *numberOfCPUs* information is extracted from the VMInstance. The CloudProvider is obtained from the Provider type in the CloudML model. The *location* attribute of the VM is acquired from the *location* attribute of the ExternalComponent type in CloudML. For the InternalComponent must be expressed which Component instances it requires: the information is

extracted from the RequiredHostInstance relationship in the CloudML instance model.

In addition, also the information about methods should be saved in the Knowledge Base to manage the monitoring activity. This is fundamental to enable the monitoring activities not only at machine level but also at application level. A different approach in this case is needed, because, as we have seen, no information about the methods is present in CloudML. To resolve this problem we decided to use the Data Collector at application level to retrieve the necessary information, since this component is capable to discover all the methods offered by its target application.

In our architecture, during the initialization phase the Data Collector calls the API of the Monitoring Manager and sends the information about Methods. Since the methods declared by the application are static and cannot change at run-time, we opted to communicate the provided methods to the Monitoring Platform during the initialization of the Data Collector, achievable thanks to the static nature of the offered methods. Originally, the relationship *providedBy* was inverted: the InternalComponent required a set of methods through the relation *providedMethods*. But this solution created inconsistencies in case of updating an InternalComponent, because CloudML does not have information about methods, so when updating the information of an InternalComponent the provided methods would be lost (*i.e.,* the Knowledge Base API, when updating, firstly deletes the instance and then re-instantiates the updated component). In this way, a method is always associated with the application and we are able to retrieve the offered methods of each application. This logical dependency is used when there is the need to remove an application from the model to delete all the offered methods. This is a key point of this part of the project since there is never a specific request to remove a method at run time. The request is always hidden in another request (*e.g.,* delete a component) because, if an application no longer exists, also the offered methods do not exist any more and so, they should be removed from the model.

4.3 Status monitor

A key factor to permit that the whole platform evolves in a right way is that the information in the models is correct and updated (see requirement R_2). As already described, CloudML updates its own model before performing any change, nevertheless, the user may perform some changes directly on the provider's console or the machines may disappear for external causes (*e.g.,* power loss, network blackout). In the original platform, there is no way for Models@Runtime engine to discover these kinds of situation. We extended the CloudML in order to improve its ability to represent the status of the deployment of running system, by monitoring the status of VMs, whilst still being independent from the Monitoring Platform. It ensures, on one hand, the casual connection between the Models@Runtime's model and the running system and keeps the model always updated, while, on the other hand, creates a stronger independence between the two components.

The component is implemented using a pull policy to detect the changes (*i.e.,* retrieving the status at a given frequency) and not using a pushes policy (*i.e.,* asking the provider for sending a notification if something changes). This is due to the fact that not all the providers offer push notifications services and sometimes they should be paid as extra (*e.g.,* Amazon SNS). In addition, the providers have a tolerance window that sometimes is in the order of minutes, while using our system the window can be set to the duration best fitting the need of the application or service deployed. Another advantage of a pull policy is that it is able to detect failures due to events that might prevent the correct working of a push policy. Examples of cases in which this could happen are the sudden power loss in a machine or network crashes. In these situations, the unreachable machine cannot communicate its status but if a request is performed, the problem can be easily detected.

Chapter 5

Implementation

"The only way to enjoy anything in this life is to earn it first."

Ginger Roberts

In this chapter, referring to the architecture described in Section 3.2 we focus our attention on how we implemented the functionalities. We will detail how the components work internally and how the interactions between them are managed. For each component, we will provide the link to the repository where the code can be found. Please note that, since the components were already existent, the repository contains also work already done by the other partners of the project.

The Deployer Engine consists in CloudML developed by Sintef and structured as a unique project with multiple sub-modules. Mainly, our contributions are located in package Monitoring that has been completely developed by us. We also modified the class CloudAppDeployer, in the package Deployer, to ensure that the monitoring components (*e.g.,* the Status Monitor) are started if requested by the user.

The Monitoring Platform developed by Politecnico di Milano is composed of many components. Each component of the Monitoring Platform is a separate project and we modified the projects relevant for our work, in particular: the Monitoring Manager, the QoS model, and the AppLevelDC.

All the projects and modules are developed using Java and they are Maven projects. Maven is a build management tool that simplifies the solution of the dependencies and grants that different modules can be reused across different

projects easily. We implemented the new package developed during our work as new Maven modules to make possible their re-usage in other parts of the system.

All the links to repositories we provide in this chapter, as references, point to branches we created to modify the components. Nevertheless, the changes have been merged also in the respective master branches and they are actually used in the MODAClouds project.

5.1 APIs

As we have already described in Chapter 4, we implemented four APIs: two for add/delete components, one to send the whole model, and one to get a component. Respecting the logic of the Monitoring Platform that offers a RESTful interface, we implemented this four new APIs as RESTful methods that can be called to interact with the model. Upon receiving the request, the information is extracted and the Monitoring Manager calls the APIs in the Knowledge Base to act on the model.

Put a model

We implemented an API to insert the whole model at the conclusion of the deployment phase. If a previous model existed, the existing model is replaced. A new resource is created for each resource in the uploaded model with the id specified.

Listing 5.1: Put model definition

```
1 Endpoint: PUT /v1/model/resources
2 Data parameters: A JSON containing the new model that must be uploaded on the
      knowledge base
3 Response: 204 No Content
4 Errors: 505 error while uploading the model - The model was not valid
```

Listing 5.1 specifies the API definition; as you can see, the API can be contacted at the address of the Monitoring Platform specifying the endpoint indicated. It requires as parameter a JSON file, which contains the resources to store in the model. In case of success, the response code is *204*, otherwise, if for instance the JSON file is not syntactically correct, the error code is *505*. As previously written, if a model is already stored in the Knowledge Base, a new PUT request completely overrides the model stored.

Insert or update resources

This API is designed to insert new resources in the Knowledge Base or update already existing resources. If a previous model existed, the existing model is updated. A new resource is created for each resource in the uploaded model with the id specified. If a resource with the specified id already exists, the resource is replaced with the new one.

Listing 5.2: Post resources definition

```
1 Endpoint: POST /v1/model/resources
2 Data parameters: A JSON containing the new model that must be uploaded on the
      knowledge base
3 Response: 204 No Content
4 Errors: 505 error while uploading the model - The model was not valid
```

Listing 5.2 specifies the API definition; as you can see, it requires as parameter a JSON file as for the PUT API. However, in this situation the model does not overwrite the model previously stored in the Knowledge Base, but just adds the new instances. If a resource already exists in the model, it will be overwritten with the new information. In case of success, the response code is *204*, otherwise, if for instance the JSON file is not syntactically correct, the error code is *505*.

Delete a resource

This API permits to delete a resource stored in the Knowledge Base that no longer exists in the Models@Runtime's engine model.

Listing 5.3: Delete resources definition

```
1 Endpoint: DELETE /v1/model/resources/:id
2 Data parameters: None
3 Response: 204 No Content
4 Errors: 404 Resource not found - The resource does not exist
5         505 Error while deleting component
```

Listing 5.3 specifies the definition of the API; as you can see the *id* of the resource must be declared directly in the URL when contacting the Monitoring Platform. No data parameter is required. The response code is *204* if the resource has been found and deleted, *404* if the resource has not been found, and *505* if an internal server error occurred.

We spent some extra attention for correctly manage in the Monitoring Manager the dependencies between components. In the model, there are dependencies

between an application and the machine that hosts it and between methods and the application that offers it. In our system, the Monitoring manager analyses the dependencies when deleting a resource and resolves them eliminating the nested components in the right order. More in detail, each InternalComponent contains the property *requiredComponents* with the IDs of the components from which it depends. Every time there is a delete request, before deleting the component, the system searches and deletes all the applications or components depending by that specific component. For example if a machine goes off-line, we take care of deleting from the Monitoring Platform all the applications that were running on that specific machine. In a similar way, each method contains a reference to the application that offers it (*i.e.,* property *providedBy*) so that, if there is the need to delete an application, also the methods are deleted. Finding a way to delete methods automatically was important. Since Models@Runtime does not have any information about the methods, we could not get any notification about them coming from the deployer as it provides for the resources.

Get a resource

The API permits to retrieve information of a specific component.

Listing 5.4: Get resources definition

```
1 Endpoint: DELETE /v1/model/resources/:id
2 Data parameters: None
3 Response: 200 Success
4 Errors: 404 Resource not found - The resource does not exist
5         505 Error while deleting component
```

Listing 5.4 specifies the definition of the API; as for the DELETE API, the *id* of the resource must be declared directly in the URL. The response code is *200* if the resource exists and, in this case, the body contains details about it, *404* if the resource has not been found, and *505* if an internal server error occurred.

Repository reference: Monitoring Manager [37].

5.2 Model translation

The implementation of the Model Translator, has been preceded by an accurate study of the two models to detect which information, contained in the Models@Runtime's model, is relevant for the Monitoring Platform. The next step was to map each element of the Models@Runtime's model on the right element in the QoS Model (*i.e.,* the model used for monitoring). As we described in Section 4.2.3, we implemented the model translator in CloudML for performance reasons.

The Model Translator retrieves the parts of the CloudML's model that are relevant for the Monitoring Platform.

More in details, we retrieve:

- **VMs**, necessary in the Monitoring Platform's model to configure correctly the Data Collectors at machine level
- **InternalComponents**, necessary in the Monitoring Platform's model to configure correctly the Data Collectors at application level
- **CloudProviders**, necessary to configure properly the Data Collectors with the information about the provider that offers the machine
- **Locations**, necessary to give a geographical location to the virtual machine

After fetching all these classes, the Model Translator needs to transform them in a format compatible with the Monitoring Platform's model. In order to do this we imported in CloudML the project QoS model that contains the definition of the classes used in the Monitoring Platform's model. The Model Translator maps the CloudML class on the QoS class (see Listing 5.5 for an example or the repository for the full implementation)

Listing 5.5: Code snippet to perform the translation of a VM

```
1  private static VM fromCloudmlToModaMP(VMInstance toTranslate) {
2          VM toReturn = new VM();
3          String id = toTranslate.getName();
4      toReturn.setId(id);
5      toReturn.setType(String.valueOf(toTranslate.getType().getName()));
6      toReturn.setCloudProvider(toTranslate.getType().getProvider().getName());
7      toReturn.setLocation(toTranslate.getType().getLocation());
8      toReturn.setNumberOfCPUs(toTranslate.getCore());
9      return toReturn;
10     }
```

After its creation, the model is serialized and sent to the Monitoring Manager.

Repository references: Model Translator [38] QoS Model [39].

5.3 Communication

The communication between the Model Translator, located in Models@Runtime engine, and the Monitoring Manager, in the Monitoring Platform, is based on APIs; this means that the two components have to deal with network communication.

To facilitate our work we used external libraries as support in this phase. As we have described, once the model is translated, it has to be serialized in a Json file to be sent to the Monitoring Platform. We performed the serialization using the Gson library [40] offered by Google as support to create the Json file and then we use Kevin Sawicki's HTTP-request [41] to manage the network level and actually send the message. In the Monitoring Manager, we use again the Gson library to de-serialize the model and then we call the methods of Knowledge Base to perform the saving.

In order to automate the communication between the two components, it is necessary that Models@Runtime knows the location of the Monitoring Platform. For this reason, we added the possibility while bootstrapping CloudML to specify the address of the Monitoring Platform. We offer this possibility through a configuration file and we wrote the code that deals with the file de-codification when Models@Runtime is started. Listing 5.6 presents an example of this file that specifies to use the Monitoring Platform hosted at 192.168.11.6 and listening on port 8170.

Listing 5.6: Monitoring configuration file

```
1 #MONITORING PLATFORM PROPERTIES
2 use=true
3 address=http://192.168.11.6:8170
```

Since one of the requirements of our work was to not change the behaviour of CloudML and Models@Runtime if the user does not need using a monitor platform, if the file is missing the system interpret this as a false value and all the functionalities described are switched off.

Repository references: Monitoring Manager [37] Update Sender [42].

5.4 Data Collector at application level

One of the biggest open points during the project was to find a good way to deal with the methods offered by applications. CloudML and Models@Runtime engine

are focused on provisioning and managing the machines. For this reason, the model they use contains information about applications and containers, to deploy them in the right order but then, at run-time, Models@Runtime does not care about the applications status and does not analyse the offered methods. Since we needed also information about the methods to configure the Monitoring Platform, we decided to send the presence of methods directly from the application level DC. We added to the DC all the code necessary to send the methods to the Monitoring Manager.

The user annotates the methods that should be monitored during the set up of the system. Listing 5.7 presents a simple example of a method annotated to be monitored. As it shows, it is sufficient to add the keyword *"@Monitor"* before the method and to specify the type. A new instance of a method with the given type is saved in the Knowledge Base

Listing 5.7: Code snippet showing an annotated method

```
1 @Monitor(resourceType = "register")
2 protected void doPost(HttpServletRequest request,HttpServletResponse response)
      throws ServletException, IOException {
3         parseReq(request, response);
4 }
```

During the configuration phase, the DC scans the code of the applications and searches for all the methods to monitor. This information has to be sent to the Monitoring Manager so that it can update the Knowledge Base. For each method the DC at application level retrieves the *type* of the method and the *ID* of the host application. To ensure that the *ID* field of the method is unique in the Knowledge Base we create it concatenating the application ID and the method type that are unique by definition. At the end of the configuration phase, the DC sends to the Monitoring Manager APIs a Json file containing all the IDs of methods and the ID of the application. In this way, the Monitoring Manager can correctly update the Knowledge Base and create a dependency between the method and the application that offers it (*i.e.,* save the application ID in the property *providedBy* of the method). As we have seen, this dependency ensures that, in case the application is deleted (*e.g.,* the machine that was hosting it is removed from the model) also, the methods it offers are removed so that the Monitoring Platform can act consistently.

A key point is to update the Knowledge Base in the right order inserting the application before inserting the methods it offers. Otherwise, the method will contain a not valid reference in the field *providedBy*. To achieve this result we used the Get API that we added in the monitoring manager. We implemented the DCs

enforcing that before sending the methods it checks if the application is present in the Knowledge Base and waiting if this is not true. This check ensures that the methods are always saved after the application so that the data in the Knowledge Base is always consistent.

Repository reference: Application Level DCs [43].

5.5 Model saving

We explained that the Knowledge Base consists of a RDF triple store (see Section 3.2.2. The Knowledge Base offers its own APIs and to change the model the Monitoring Manager performs requests to these methods.

The APIs that we use to manage the resources in the models are:

1. **Add resources:** it requires a set of resources as parameter and the name of the model where to store the data. The result is that the resources are stored in the Knowledge Base. This method is called both when posting new resources in the Knowledge Base and when putting a new model.
2. **Get resource by *id*:** it retrieves the component with the specific *id* passed as parameter. It is used when getting the information of a resource and when deleting it, because in case the resource is not found during the removal, the request's response code is *404*.
3. **Get all IDs:** it retrieves all the IDs of a specific type of class passed as parameter. It is used when putting a new model in the Knowledge Base so that the manager retrieves all the resource and then deletes them to overwrite the new model.
4. **Delete resource by property:** it deletes resources according to the value or values passed as parameter. It is a very useful method that automatically pinpoints the resources that have the specified property value as attribute and deletes them. It is used before the saving of a complete new model, to delete all the pre-existent resources. It is used also during the delete of a resource, especially when eliminating a component with dependencies, because it is possible to delete the related components searching them by the *id* of the root component to remove.

5.6 Status Monitor

We added the Status Monitor as component of CloudML; this component retrieves the statuses of machines from the various providers and updates the CloudML's model when they change. More in detail, the Status Monitor instantiates a communication channel for each provider, used in the deployment phase. The Status Monitor is deeply integrated with the package Deployer and it retrieves the credentials automatically. As soon as the deployment is complete, the Status Monitor starts automatically and collects information about the status of the virtual machines using the retrieved credentials. It has an internal buffer to save the status of each machine and if a change occurs between two consecutive checks it updates the model with the new status. We decided to create this internal memory to propagate the detected status to the rest of the system only in case of changes and not at each check. The update of the model is done using CloudML internal APIs and, since an internal notification service is available, this change can be propagated easily to all the components that are monitoring the model.

To ensure the complete independence between the Status Monitor and the rest of the system it runs in a separate thread. This makes also easy to extract and reuse the code in different contexts. In addition, this component is implemented in a way that it can be configured by the user through a configuration file that specifies if the checks should be done and their frequency. In the example the file requires to use the Status Monitor and to check for changes every 60 seconds (see Listing 5.8).

Listing 5.8: Status Monitor's configuration file

```
1 #STATUS MONITOR PROPERTIES
2 activated=true
3 frequency=60
```

Once again, since this is something that we have added to Models@Runtime, if the file is missing the system simply does not use the Status Monitor (*i.e.,* interprets it as "activated=false").

Repository reference: Status Monitor [44].

5.7 Observer in CloudML

It is possible to retrieve monitoring data from the Monitoring Platform implementing an observer and registering it in the Monitoring Manager (see Section 3.2.2).

Enriching the Models@Runtime's model with high-level information coming from the Monitoring Platform (*e.g.,* average CPU usage) enables the possibility to perform reasoning and adaptation directly in CloudML. In order to create this enriched model we imported a Metric Observer in the Models@Runtime engine, in this way, it is possible to receive the stream of metric values. The Models@Runtime's model does not have fields to save this information but it is possible to add to any object in the model a property. A property is a couple "key" and "value" that can be added to any component. In this way, the monitored values can be associated with the machine they refer to. Another advantage of importing this information is that future releases of Models@Runtime can utilize the monitored data to react to particular situation. We have done some tests of this mechanism and it works but, at the moment, it is not used because, due the current early stage of CloudML, there are no advantages in knowing the machine performances at run-time.

Repository reference: Imported Observer [45].

Chapter 6

Evaluation

"The strongest arguments prove nothing so long as the conclusions are not verified by experience. Experimental science is the queen of sciences and the goal of all speculation."

Roger Bacon

In this chapter, we reuse the motivating example presented in Section 3.4 evaluating how the solution described in Chapter 4 impacts on the case study. Then, we will evaluate the performances of the new architecture and how it satisfies the requirements described in Section 3.5.

The goals of this evaluation are:

- Check if the requirements are fulfilled
- Evaluate the impact of our contribution on the performances of the system
- Evaluate the performances of the system on different cloud providers

6.1 Case study

The goal is to deploy MIC (see Section 3.4 in a multi-cloud environment adapting the monitoring activity according to the evolution of the running system. The application is the same of the motivating example, with only one difference: we included in the frontend application the library that permits to collect data also at application level.

Initially, the deployment is hosted on a single cloud provider. The Models@Runtime engine creates the VM and installs the software declared in the model provided by the user. With the introduction of our contribution the Models@Runtime engine, using the communication channel, can send the model of the system to the Monitoring Platform as soon as the deployment is completed (see Figure 6.1).

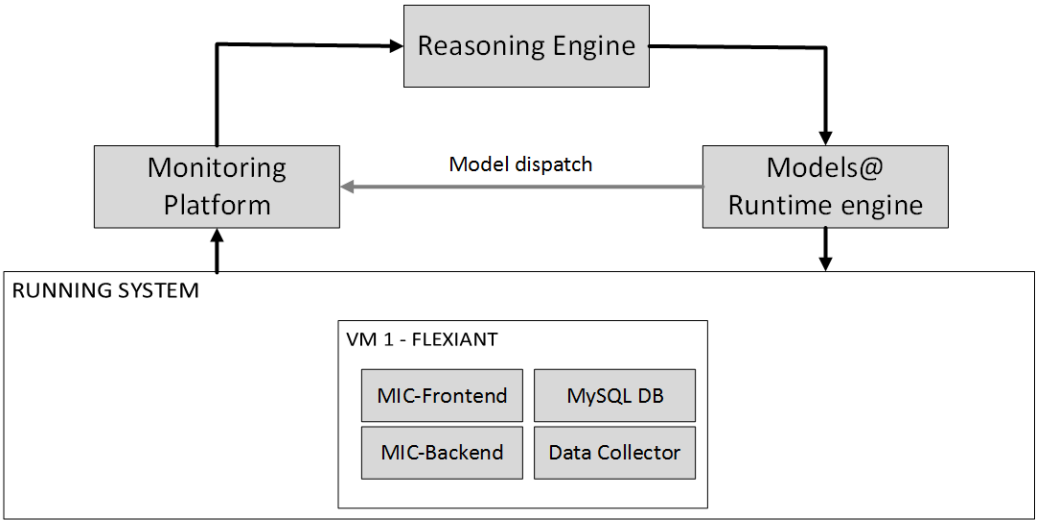

Figure 6.1: Dispatch of the model at the conclusion of the deployment phase

The Model Translator in the Models@Runtime engine translates the model in a format compatible with the Monitoring Platform, and then uses the Put API to send the model (see Listing 6.1) to the Monitoring Manager. When the model is sent to the Monitoring Manager, it is stored in the Knowledge Base.

Listing 6.1: The Json file sent by the Model Translator at the end of the first deployment

```
{"cloudProviders": [{
    "id": "Flexiant",
    "type": "Flexiant"
  },{
    "id": "Ec2",
    "type":"Ec2"
  }],
 "internalComponents": [{
    "id": "Mic-Frontend",
    "requiredComponents": ["VM1"],
    "type": "mic-frontend"
  },{
```

```
13        "id": "Mic-Backend",
14        "requiredComponents": ["VM1"],
15        "type": "mic-backend"
16      },{
17        "id": "Glassfish",
18        "requiredComponents": ["VM1"],
19        "type": "glassfish"
20      },{
21        "id": "Memcached",
22        "requiredComponents": ["VM1"],
23        "type": "memcached"
24      },{
25        "id": "MySQL",
26        "requiredComponents": ["VM1"],
27        "type": "mysql"
28      }],
29    "vMs": [{
30        "cloudProvider": "Flexiant",
31        "id": "VM1",
32        "numberOfCPUs": 1,
33        "type": "MIC-Node"
34      }]}
```

Note that all the cloud providers must be declared in the first deployment model of the application. The Models@Runtime engine reports the log of the dispatch of the request, with the names of the components added and the result of the request (see Listing 6.2).

Listing 6.2: CloudML's log reporting the successful communication

```
1  >> Connecting to the monitoring platform at http://109.231.122.205:8170
2  >> Sending VM: VM1
3  >> Sending Provider: Flexiant
4  >> Sending Provider: Ec2
5  >> Sending InternalComponent: MIC-Frontend
6  >> Sending InternalComponent: MIC-Backend
7  >> Sending InternalComponent: Glassfish
8  >> Sending InternalComponent: Memcached
9  >> Sending InternalComponent: MySQL
10 >> Connection result: 204 Request successful, no content
```

In parallel, the Application Level DC gathers the annotated methods found in the code that have to be sent to the Monitoring Manager. Listing 6.3 specifies the annotated methods in the application. Note that the types are coherently declared according to the monitoring rule specified in Listing 3.2 and this permits a proper monitoring of the methods.

Before sending the methods, the Application Level DC checks if the *applicationId* of the InternalComponent that offers the method is present in the Knowledge Base through the Get API; if the component is not present, it waits for the completion of the deployment. This ensures that the information about methods is saved in the Knowledge Base only if it is updated and consistent, avoiding the insertion of erroneous or partial data.

Listing 6.3: The Json file sent by the app-level-dc at the end of the first deployment

```
{ "methods":[   {
    "id":"Mic-frontend-answerQuestions",
    "type":"answerQuestions",
    "providedBy":"Mic-frontend"
  },{
    "id":"Mic-frontend-saveAnswers",
    "type":"saveAnswers",
    "providedBy":"Mic-frontend"
  },{
    "id":"Mic-frontend-register",
    "type":"register",
    "providedBy":"Mic-frontend"
  }]}
```

In the next step, we check if the components are properly stored in the Knowledge Base requesting them through the Get API. As you can see in the outcome messages reported in Listing 6.4, each component is correctly stored in the Knowledge Base. The attributes for each component are correctly assigned according to the Json file of the model specified in Listing 6.1.

Listing 6.4: HTTP requests that check the stored components

```
GET /v1/model/resources/VM1
Body response: VM [numberOfCPUs=1, location=, getCloudProvider()=Flexiant,
    getType()=MIC-Node, getId()=VM1]
GET /v1/model/resources/MIC-Frontend
Body response: InternalComponent [requiredComponents=[VM1], getType()=mic-
    frontend, getId()=MIC-Frontend]
GET /v1/model/resources/MIC-Backend
Body response: InternalComponent [requiredComponents=[VM1], getType()=mic-backend
    , getId()=MIC-Backend]
GET /v1/model/resources/MIC-Frontend-saveAnswers
BodyResponse: Method [providedBy=MIC-Frontend, getType()=saveAnswers, getId()=MIC
    -Fronted-saveAnswers]
```

At this point, the Monitoring Platform's model is coherent with the running system, it contains all the deployed resources (see Figure 6.2) and the Data Collectors are installed on the machine; to start the monitoring, only the monitoring rules are missing. It is possible to install the rules though the Monitoring Manager's interface. After the installation, a query is generated from the rule and then sent to the Data Analyzer and a configuration file is saved in the Knowledge Base. The DCs refresh their configuration downloading the new file and the monitoring starts.

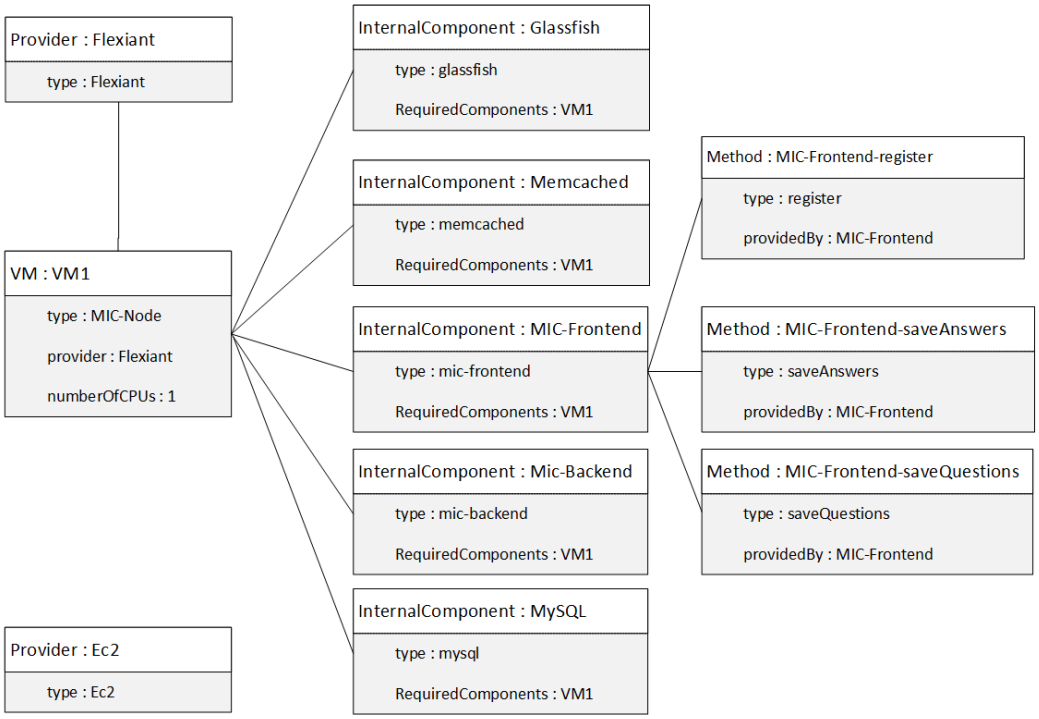

Figure 6.2: Model stored in the Knowledge Base at the end of first deployment

Now we can check if the DC at infrastructural level retrieves correctly its configuration from the Knowledge Base. As shown in Listing 6.5, the DCs find the respective data collector generated by the monitoring rules in the Knowledge Base. They download their configuration with the parameters to monitor properly the resources, one to monitor the VMs and the other to monitor the methods. Then, they start sending monitored values to the DA.

Listing 6.5: Data Collector's log showing the synchronisation with the Knowledge Base

```
1  - Syncing with KB..
2  - 2 data collectors were downloaded from KB
3  - Resource VM1 required to be monitored according to DCMetaData [id=dc
       -120574383, monitoredMetric=CpuUtilization, parameters={samplingTime=10,
       samplingProbability=1}, monitoredResourceClasses=[VM],
       monitoredResourcesTypes=[MIC-Node], monitoringRulesId=mr_1]
4  - Resource VM1 required to be monitored according to DCMetaData [id=dc1106058117
       , monitoredMetric=ResponseTime, parameters={samplingProbability=1},
       monitoredResourceClasses=[Method], monitoredResourcesTypes=[answerQuestion,
        saveAnswers, register], monitoringRulesId=respTimeRule]
5  - Data collectors synced with KB.
6  - Sending datum 0.145368483105834 CpuUtilization VM1
```

The Data Analyzer receives the monitored data and aggregates them according to the rule. Listing 6.6 illustrates the reception of the monitoring datum sent by the DC. The Post request to the Monitoring Platform at the port of the DA has response code of *200* and this means that it has been correctly elaborated. More in details, the DA receives the data coming from the VM and then processes them to offer the metric flow at the output.

Listing 6.6: Data Analyser's log showing the synchronisation with the DC

```
1  - 109.231.121.44 8175 POST /streams/http://www.modaclouds.eu/streams/
       cpuutilization - 200
```

At this point, following the example, it is supposed that the Reasoning Engine notices that the CPU utilisation is too high and it decides to modify the deployment moving the frontend application to another VM.

Using the communication channel, it is possible to send the updates to the Monitoring Platform (see Figure 6.3). Note that there are no more the dashed arrow and the question marks of Figure 3.8, because now the Monitoring Platform is aware of the changes in the deployment.

The Models@Runtime's engine receives the target deployment model provided by the Reasoning Engine and computes the difference between the current model and the target one. It analyses which operations have to be performed in order to adapt the deployment model and acts on the running system.

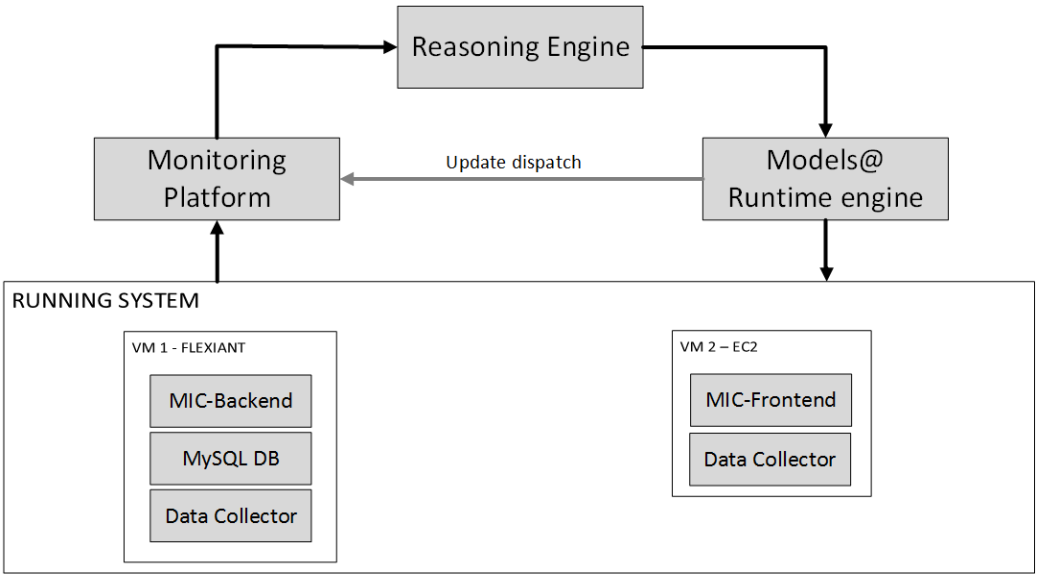

Figure 6.3: Dispatch of the model updates at the conclusion of the migration

Listing 6.7: Output of the difference between the current model and the target model

```
1 >> Comparing VMs ...
2 >> Removed components : []
3 >> Added components : [Instance VM2 : MIC-Node]
4 >> Comparing Internal Components ...
5 >> Removed components : [Instance MIC-Frontend : mic-frontend, Instance Memcached
      : memcached]
6 >> Added components : [Instance MIC-Frontend2 : mic-frontend, Instance Memcached2
      : memcached, Instance Glassfish2 : glassfish]
```

After computing the difference (see Listing 6.7), the Models@Runtime engine provisions the new machine, installs the new software, and then sends the model update to the Monitoring Platform contacting the Post API (see Listing 6.8). As described, the Post request does not override the model contained in the Knowledge Base, but just adds the components to the model.

Listing 6.8: The Json file sent by the Model Translator at the end of the migration

```
1 {  "locations":[    {
2       "id":"eu-west-1",
3       "type":"eu-west-1"
4     }    ],
5   "internalComponents": [    {
6       "id": "Mic-Frontend2",
7       "requiredComponents": ["VM2"],
8       "type": "mic-frontend"
```

```
9       },{
10        "id": "Glassfish2",
11        "requiredComponents": ["VM2"],
12        "type": "glassfish"
13      },{
14        "id": "Memcached2",
15        "requiredComponents": ["VM2"],
16        "type": "memcached"
17      }],
18    "vMs": [{
19        "cloudProvider": "ec2",
20        "id": "VM2",
21        "location":"eu-west-1",
22        "numberOfCPUs": 1,
23        "type": "MIC-Node"
24      }]}
```

The CloudML's log, reported in the Listing 6.9, shows the sequence of the operations performed: it deletes the components no longer present in the deployment model and then it sends the updates to the Monitoring Platform.

Listing 6.9: CloudML's log reporting the successful communication

```
1  >> Connecting to the monitoring platform at http://109.231.122.205:8170
2  >> Deleting : MIC-Frontend
3  >> Connection result: 204 Request successful, no content
4  >> Connecting to the monitoring platform at http://109.231.122.205:8170
5  >> Deleting : Memcached
6  >> Connection result: 204 Request successful, no content
7  >> Connecting to the monitoring platform at http://109.231.122.205:8170
8  >> Sending VM : VM2
9  >> Sending Location : eu-west-1
10 >> Sending InternalComponent: MIC-Frontend2
11 >> Sending InternalComponent: Memcached2
12 >> Sending InternalComponent: Glassfish2
13 >> Connection result: 204 Request successful, no content
```

The removal of the InternalComponents affects also the methods associated to them; when removing the MIC-Frontend also the methods *saveAnswers*, *register* and *answerQuestion* are deleted. The Data Collector deployed with the new instance of the MIC-Frontend sends the methods that are associated with this new instance (MIC-Frontend2). The model is still consistent (see Figure 6.4).

The monitoring rules permit to define the type of the resources to monitor, so the monitoring activity restarts automatically on the new resources with the same rules already specified, because the types of the resources are the same. Since the model stored in the Knowledge Base is consistent with the real status of the running system, the DCs can properly retrieve their configuration in order to

start monitoring the new resources. Therefore, the new architecture of the system permits to auto-adapt the monitoring activity with respect to the deployment model, achieving the possibility to perform structural adaptation at run-time.

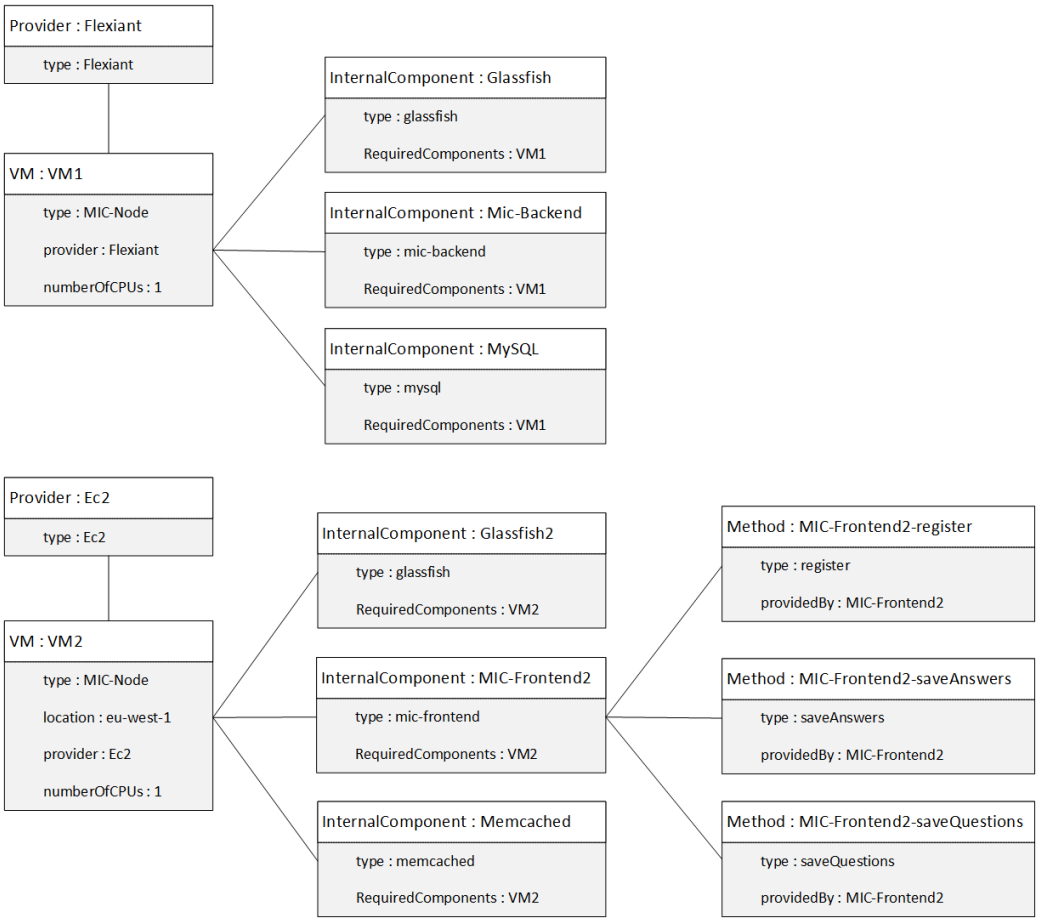

Figure 6.4: Model stored in the Knowledge Base after the migration

In case of suspension of a machine due to external accidents (third points of the example) or user's fault, the Models@Runtime engine must be capable of updating its model, noting the status of the machine changed in the deployment. To verify this case in our work, we simulated it stopping the machine VM1 from the Flexiant console, so that it becomes no longer contactable. The running CloudML instance detects the change of status, checking at the frequency set with the configuration file the status of all the machines belonging to the model stored (see Listing 6.10).

Listing 6.10: Output of the Status Monitor showing the new status of the machine

```
1 >> Looking for status changes..
2 >> VM : VM1 changed in status STOPPED
```

The Status Monitor has detected the new status of the machine and it updates the model with the new information. All the observers registered to the model receive a notification so the Reasoning Engine, if it is registered, can handle the exception.

6.2 Performances evaluation

In this section, we analyse the performances of the platform after our contribution. As first step, we will summarise all the phases of the structural adaptation performed in the case study, and later we will draw the conclusion about the impacts of our work on the system.

The proposed system, which includes an up-to-date representation of the running system, permits, when the deployment of the running system is modified (*e.g.,* migrated from one provider to another), to automatically adapt the monitoring activity keeping the monitoring running with the same settings and rules used before.

The interface provided by the Models@Runtime engine enables modifications on the deployment of the running system by uploading the desired model of the deployment. For instance, referring to our example (the migration of MIC-Frontend from Flexiant to Amazon EC2) and to Figure 6.5, the Reasoning Engine specifies the desired deployment model and provides it to the Models@Runtime engine. The internal logic compares the given model with the current one and the Models@Runtime engine operates on the running system to fulfil the requests (see Section 3.2.1). First, the Models@Runtime engine instantiates a new machine and deploys MIC-Frontend on it and then it deploys the Data Collectors on the VM. At this stage, the Models@Runtime engine notifies to the Monitoring Manager the changes in the deployment and the Monitoring Manager uses this information to update autonomously the Knowledge Base from which the Data Collector retrieves its own configuration.

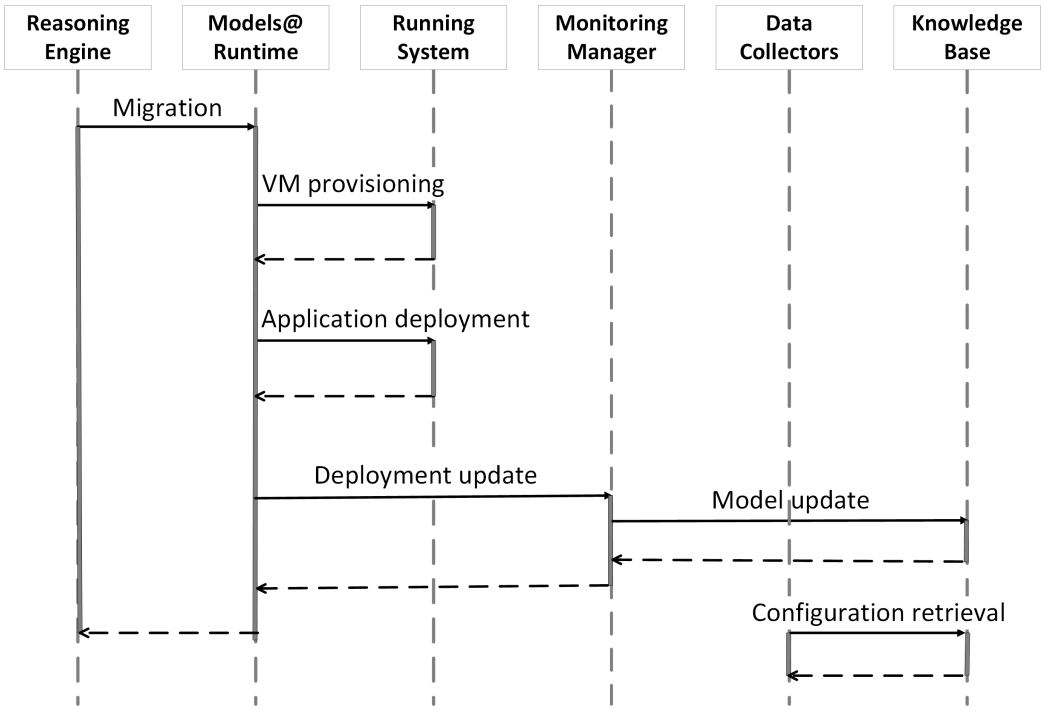

Figure 6.5: Sequence diagram of the migration of the application

The time interval between the start of migration and the adaptation of the monitoring activity mainly depends on the time needed by the Models@Runtime engine to instantiate the new machine and deploy the applications. It depends on the target model provided by the Reasoning Engine; if the actions to adapt the deployment are complex (*e.g.,* provisioning of multiple machines) the model update will take longer. Moreover, it depends also from the user specified provider and by the traffic that the provider has to manage therefore it is independent of our contribution.

We observed that, in general, Amazon ensures better performances than Flexiant, at least using CloudML. We also noticed that, the performances offered by Flexiant vary a lot due to the small dimension of the cluster we use which shared among multiple European projects.

We report the times to complete the deployment on different providers in different situations to give to the reader an idea of system's performances. The following data are extracted from the log file generated by CloudML during the deployment.

Provisioning of a virtual machine with Ubuntu 14.04 without any other applications on Amazon takes from a minimum of 1 minute and 10 seconds to a maximum of 1 minutes and 25 seconds. The same operation on Flexiant takes at least 2 minutes and 30 seconds arriving also to 4 minutes and 40 seconds in the worst case.

A complete deployment of MIC takes around 45 minutes on Flexiant as the download speed of the resources and of the packages from apt-get is never more than 60 kb/s). This time includes the provisioning of the virtual machine, the installation of Ubuntu 14.04, the download and installation of the MIC's dependencies (*i.e.,* Java, Glassfish, Memcached), the download and set up of MIC-Frontend and MIC-Backend. On the other side, Ec2 takes around 35 minutes to deploy and install completely MIC.

Every time the Models@Runtime engine finishes the operations on the running system, the updates are sent to the Monitoring Platform and it updates the Knowledge Base's model. We consider a delay from one to two seconds for each API call. Before that the monitoring activity adapts itself, there is a further delay due to the synchronisation rate defined for the Data Collectors to download their configuration from the Knowledge Base (the default value is 10 seconds). The total time to react to a change is: the time to adapt the deployment, plus the time to communicate the changes, plus the time to synchronise the Data Collectors.

Considering that:

- The time to adapt the deployment (*i.e.,* machine provisioning and software installation) is not affected by our contribution
- The frequency at which the DCs retrieve their configuration files can be tuned by the user
- The update of the model requires, in average, a small number of calls to the APIs

We could assert that the performances of the system are not reduced by our work. To help the reader to realise this, Figure 6.6 offers a visual comparison.

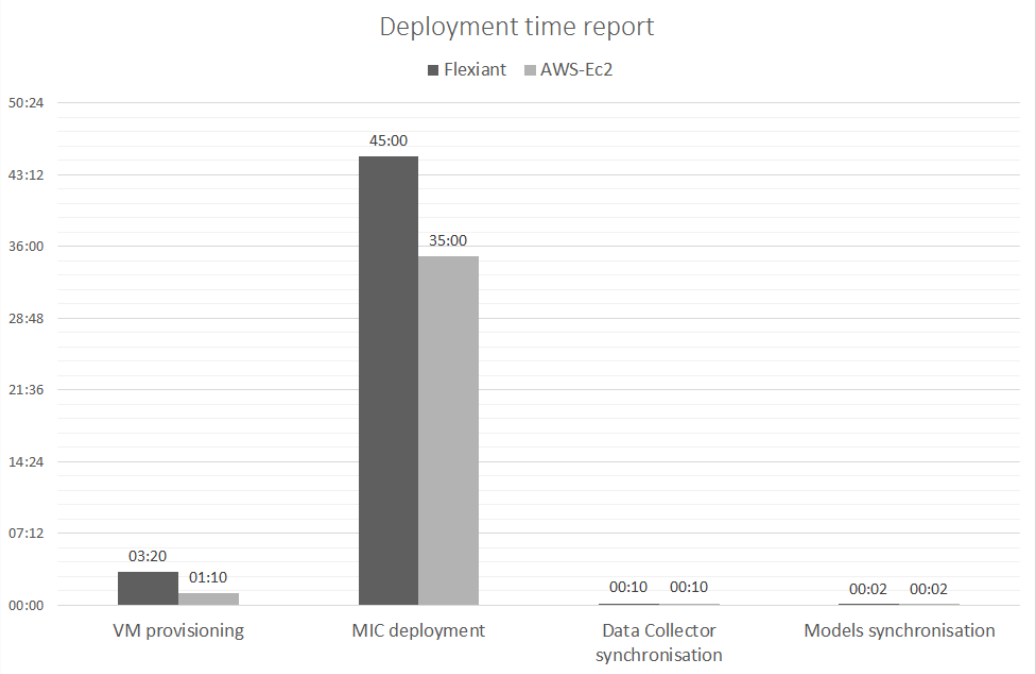

Figure 6.6: Comparison between deployment times on Flexiant and Ec2

It is possible to observe that the provision of the machine and the download, the configuration, and the installation of all the software needed are the operations that mainly affect the total deployment time. These operations are not affected by our contribution, but depend on the complexity of the system to deploy and on the cloud provider's performances. Also for a simple application like MIC, the waiting time for the installation is considerably long. Our work just adds to the system a delay of few seconds to synchronise the two models. Our conclusion is that our contribution does not affect the performances of the platform because the introduced delay is shorter by orders of magnitude than the time to act on the running system.

6.3 Fulfilling the requirements

In this section, we show that the requirements of Section 3.5 are fulfilled by the architecture proposed.

- **Cloud provider-independence (R_1)**

  The architecture proposed is cloud provider-independent for several reasons. The whole MODAClouds project is designed and implemented with the target of offering a cloud agnostic system capable of managing multi-cloud applications. Consequently, all the components included in the project are designed for a multi-cloud environment. The Models@Runtime engine permits to deploy and instantiate machines on different cloud providers. The DCs collect data independently from the cloud provider that hosts the VM. Our contribution does not affect the behaviour of these components, so it keeps on to be cloud provider-independent. The Status Monitor is implemented considering the fact that it has to monitor resources offered by different providers and reuses the connectors already present in CloudML to deal with the various providers. For these reasons, also this component can manage multiple providers.

- **Model abstraction and coherence (R_2)**

  The Models@Runtime engine stores correctly the model provided by the Reasoning Engine or by the user, and acts consequently on the running system to achieve that model. This ensures the coherence between the given model and the reality. Moreover, the introduction of the Status Monitor permits to have the run-time status of the VMs that before was not available. After the introduction of our contribution, the model is always consistent with the reality also at run-time, in case some external accidents or user's faults modify the status of a machine. Furthermore, the translation from the Models@Runtime engine's model to the Monitoring Platform's one permits to have the exact abstract representation also in the Knowledge Base: this is fundamental to fulfil the following requirements.

- **Co-evolution of the monitoring processes with the application (R_3)**

  Thanks to the communication channel created between the Monitoring Platform and the Models@Runtime engine, it is possible to co-evolve the monitoring processes with the run-time application. The Models@Runtime engine communicates from scratch the first deployment model at the start, and then if any modification occurs updates the Monitoring Platform model advising about the changes enacted. Exploiting the API to put a model, update resources and delete resources, the Models@Runtime engine can manage every kind of adaptation of the model stored in the Knowledge Base.

- **Adaptation of the monitoring platform at run-time (R_4)**

  In order to adapt dynamically the monitoring activity at run-time, the DCs have to retrieve correctly their configuration form the Knowledge Base. To achieve this target, our work must satisfy two constraints: the Models@Runtime engine must properly send the deployment model and its updates with the correct information for each resource, and the storing of the resources must be handled in the right way when receiving the communications from the Models@Runtime engine. As already said for the requirement (R_2), the Models@Runtime engine sends the information about each resource when transmitting the model. The Monitoring Platform correctly handles the requests coming from the Models@Runtime engine and stores accurately the resources in the Knowledge Base, respecting also the dependencies among the components. This ensures that, as demonstrated in the case study, the DCs can precisely retrieve their configuration and manage the adaptation of the monitoring activity at run-time.

Chapter 7

Conclusions and future research

"If everyone is thinking alike, then somebody isn't thinking."

George S. Patton

As illustrated through this document, the work is positioned in the context of multi-cloud applications. More in details, this master thesis was conducted in the context of the European project MODAClouds that aims to create tools and methodologies for designing, deploying and managing easily multi-clouds applications. The goal of our work was to enable the dynamic adaptation of the Monitoring Platform activity according to the changes performed by the Models@Runtime in the deployment. With our contribution, the monitoring activity is able co-evolve with the system changes at run-time and always be consistent with the status of the deployment.

7.1 Current status

In Section 3.5, we identified some requirements that we would have to fulfil to ensure a high quality and actual useful platform. Such requirements are not specific of this project but, in our opinion, every good system to manage multi-cloud applications should fulfil them.

As demonstrated from the case study in Section 6.1, the platform is provider independent because both the Monitoring Platform and the Models@Runtime engine work indifferently with multiple cloud providers. Both components rely on

a provider-agnostic model with domain-specific concepts. The components added have been designed to maintain the cloud provider independence avoiding the user to incur in the vendor lock-in problem (R_1).

Using model-driven engineering, our platform provides an abstract representation of both the running system and the monitoring processes. The model-driven engineering permits to react to each modification occurred in the running system. Communicating the modifications of the deployment at run-time to the Monitoring Platform, the consistency of the model stored in the Knowledge Base with the reality is granted (R_2). Furthermore, the Models@Runtime engine can retrieve the status of the virtual machines autonomously.

The Monitoring Platform offers all the needed interfaces to communicate any kind of change occurred in the Models@Runtime's engine model, permitting the co-evolution of the monitoring activity with the application deployment (R_3).

In the same way, the Monitoring Platform can dynamically adapt itself properly configuring the Data Collectors and the Data Analyser. In addition, if something is changed in the deployment, this information is automatically sent to the Monitoring Platform by the Models@Runtime engine, without the need for the user or third parties entities to reconfigure the monitoring activity (R_4).

As shown, choosing the right components, and connecting them with the communication channel, ensures that the platform fulfils the requirements. In our opinion, this makes our system actually useful and capable to deal with the deployment and monitoring of complex systems in a multi-clouds environment.

Examples of its capabilities have been shown in Chapter 6 in which we used it in a real case scenario. The requirements imposed permit to extend the results achieved also to situations different from the scenario presented, giving the possibility to adapt the monitoring activity in environments that involve other cloud providers and other applications.

Moreover, since our work on the system consists in a set of modules, it is easy to extend the platform to add other functionalities and the already present features can be extended in a simple way.

Finally, from the analysis of the performances illustrated in Section 6.2, we can conclude that the platform proposed does the performances of the system during the adaptation of the monitoring activity.

7.2 Future work

The goal of this work was to synchronize the Deployer and the Monitoring Platform to ensure that the models were always consistent with the status of the running system. This permits to adapting the monitoring according to the changes in the running system. Using this extended architecture further paths can be explored and new functionalities can be added.

7.2.1 Add more Data Collectors

At the moment, the project supports a Data Collector at machine level and a Data Collector at application level. Using these Data Collectors, it is possible to monitor the performances of the machine (*e.g.,* CPU usage) and to monitor Java methods. A future work could focus on (and in the context of MODAClouds this probably will happen) adding more data collectors at container or application level to retrieve further information. If more Data Collectors will be added, it will be necessary to analyse them and detect if, to properly work, they need some information that, at the moment, is not synchronised between the two models. If this would be the case, this information might be exchanged using the communication channel already present in the system.

7.2.2 Create an enriched model

As we said, we introduced the communicational channel, mostly, to update the Monitoring Platform when something changes in the deployment. We have enabled the possibility to receive the monitoring data in CloudML but we do not use it. Enriching the model in the Models@Runtime with monitoring data is possible to open the door to many further possibilities.

First of all, external entities (*e.g.,* a reasoning engine, a user...), with the observation of only this model, can get information about the status of the system at run-time, retrieved by the Monitoring Platform, only interacting with Models@Runtime. In addition, it would be possible to plug stateless (*i.e.,* without a system representation) external entities using this unique enriched model to take decisions on the system.

Moreover, it is possible to build a system that automatically deploys a monitoring platform together with the applications and installs a set of "basic" monitoring

rules (*e.g.,* CPU usage, number of disk operations...). A system built in this way would offer all the facilities to start, manage, and monitor multi-clouds applications simply and in a very fast way. An important aspect of doing something like that is that the components can be used as they are, without the need of big changes. The only required work would be to deploy the Monitoring Platform together with the system and then create the monitoring rules. Both these two operations could be performed by the Models@Runtime that is capable of deploying the Monitoring Platform and of sending commands (*i.e.,* install monitoring rules).

During our work, we have designed an architecture similar to this one "with the Monitoring Platform hidden from the final user" and with a "big model" but then we have excluded it because the reasoning engine that is being developed in the MODAClouds project will work with the two components independently. A problem that this future work should deal with, is that the creation of a "unique big model", to which all the components refer to, introduces a single point of failure in the system. A possible way to mitigate this would be the duplication of the model with the possibility of "hot swapping" them when a failure occurs.

Bibliography

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, Special Publication 800-145, September 2001.

[2] D. Petcu, "Consuming resources and services from multiple clouds," *Journal of Grid Computing*, pp. 1–25, 2014.

[3] D. Ardagna, E. Di Nitto, G. Casale, D. Pectu, P. Mohagheghi, S. Mosser, P. Matthews, A. Gericke, C. Balligny, F. D'Andria, C.-S. Nechifor, and C. Sheridan, "MODACLOUDS, A Model-Driven Approach for the Design and Execution of Applications on Multiple Clouds," in *ICSE MiSE: International Workshop on Modelling in Software Engineering*. IEEE/ACM, 2012, pp. 50–56.

[4] X. Zhang1, X. Chen, Y. Zhang, Y. Wu, G. Huang, and Q. Lin, "Runtime Model Based Management of Diverse Cloud Resources," *Model-Driven Engineering Languages and Systems*, pp. 572–588, 2013.

[5] "Brooklyn," http://brooklyn.incubator.apache.org.

[6] D. Trihinas, G. Pallis, and M. D. Dikaiakos, "JCatascopia: Monitoring Elastically Adaptive Applications in the Cloud," http://linc.ucy.ac.cy/publications/pdfs/2014_ccgrid14.pdf.

[7] D. C. Schmidt, "Model-driven engineering," *Computer-IEEE Computer Society-*, vol. 39, no. 2, p. 25, 2006.

[8] N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg, "Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud

systems," in *CLOUD 2013: IEEE 6th International Conference on Cloud Computing*. IEEE Computer Society, 2013, pp. 887–894.

[9] "Amazon Web Services," http://aws.amazon.com/.

[10] "Cloudwatch," http://aws.amazon.com/cloudwatch/.

[11] "AWS Elastic Beanstalk," http://aws.amazon.com/elasticbeanstalk/?nc2=h_l3_dm/.

[12] "Amazon Simple Notification Service," http://aws.amazon.com/sns/.

[13] "AWS Total Cost of Ownership Calculator," https://awstcocalculator.com/.

[14] "Rackspace Monitor," http://www.rackspace.com/cloud/monitoring/.

[15] "Flexiant," http://www.flexiant.com/.

[16] "Flexiant Triggers," http://www.flexiant.com/flexiant-cloud-orchestrator/.

[17] "OpenStack," http://www.openstack.org/.

[18] "Hyperic," http://www.hyperic.com/.

[19] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg, "Models@Run.time to Support Dynamic Adaptation," *IEEE Computer*, vol. 42, no. 10, pp. 44–51, 2009.

[20] G. Blair, N. Bencomo, and R. France, "Models@run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[21] "Google App Engine," https://cloud.google.com/appengine/.

[22] "Microsoft Azure," http://azure.microsoft.com/.

[23] C. Zeginis, K. Kritikos, P. Garefalakis, K. Konsolaki, K. Magoutis, and D. Plexousakis, "Towards Cross-Layer Monitoring of Multi-Cloud Service-Based Applications," pp. 188–195, 2013.

[24] "CloudStatus," http://www.hyperic.com/products/cloud-status-monitoring/.

[25] "Hyperic HQ," http://www.hyperic.com/products/real-time-monitoring/.

[26] "JClouds," http://www.jclouds.org/.

[27] "SeaClouds," http://www.seaclouds-project.eu/home.html.

[28] G. Casale, W. Wang, M. Miglierina, and V. I. Munteanu, "Monitoring Platform Final Release," http://www.modaclouds.eu/wp-content/uploads/2012/09/MODAClouds_D6.3.2_MonitoringPlatformFinalRelease.pdf, April 2014.

[29] "MODAClouds EU project," http://www.modaclouds.eu/.

[30] "Apache Jena Fuseki," http://jena.apache.org/documentation/.

[31] "C-SPARQL Engine," https://github.com/deib-polimi/rsp-services-csparql/.

[32] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng, "Composing adaptive software," *Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[33] "GlassFish Server," https://glassfish.java.net/index.html.

[34] "Memcached," http://memcached.org/.

[35] C. Atkinson and T. Kühne, "Rearchitecting the UML infrastructure," *ACM Transactions on Modeling and Computer Simulation*, vol. 12, no. 4, pp. 290–321, 2002.

[36] T. Kühne, "Matters of (meta-)modeling," *Software and Systems Modeling*, vol. 5, no. 4, pp. 369–385, 2006.

[37] "DEIB Polimi - Monitoring Manager," https://github.com/deib-polimi/modaclouds-monitoring-manager/tree/m%40r-synch.

[38] "CloudML - Model Translator," https://github.com/SINTEF-9012/cloudml/blob/monitoring/monitoring/src/main/java/org/cloudml/monitoring/synchronization/Filter.java.

[39] "DEIB Polimi - QoS model," https://github.com/deib-polimi/modaclouds-qos-models/tree/Method-insertion.

[40] "Google Gson," https://code.google.com/p/google-gson/.

[41] "Kevin Sawicki http-request," https://github.com/kevinsawicki/http-request.

[42] “CloudML - Updates Sender,” https://github.com/SINTEF-9012/cloudml/blob/monitoring/monitoring/src/main/java/org/cloudml/monitoring/synchronization/MonitoringSynch.java.

[43] “DEIB Polimi - Application level Data Collector,” https://github.com/deib-polimi/modaclouds-app-level-dc/tree/method-insert.

[44] “CloudML - Status Monitor package,” https://github.com/SINTEF-9012/cloudml/tree/monitoring/monitoring/src/main/java/org/cloudml/monitoring/status.

[45] “CloudML - Observer,” https://github.com/SINTEF-9012/cloudml/tree/monitoring/monitoring/src/main/java/org/cloudml/monitoring/metrics_observer/instance.