POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

MONITORING MODERN DISTRIBUTED SOFTWARE APPLICATIONS: CHALLENGES AND SOLUTIONS

Doctoral Dissertation of:
Marco Miglierina

Supervisor:
Prof. Elisabetta Di Nitto

Tutor:
Prof. Carlo Ghezzi

The Chair of the Doctoral Program:
Prof. Andrea Bonarini

2016 – XXVIII

Abstract

THE advent of cloud computing brought a huge change in the software release cycle. It triggered an exponential shift towards paradigms where hardware could be treated just like software, *i.e.*, accessible via API calls. Resources, such as servers or network gateways, became volatile artifacts. Deployments procedures, which could previously only be performed manually, became fully automatable. This brought an increasing number of companies to release faster and respond to market demand at an unprecedented rate.

This thesis tries, first, to highlight why monitoring is an essential part of the software release cycle and why it should be considered a first class citizen, even more so in the aforementioned context. It highlights what we believe to be the most important challenges that monitoring tools developers have to address today that were not addressed by established monitoring tools that had been used for years. These challenges are addressed by our approach and runtime platform, Tower 4Clouds, a multi-cloud monitoring platform developed as part of this thesis. The solution mainly focuses on configuring the runtime monitoring since the first design phases, starting from Quality of Service (QoS) requirements definition on top of a provider independent model of the software system. The runtime platform is able to cope with the heterogeneity and the ephemerality of the resources being monitored. With Tower 4Clouds, the user is able to define the QoS requirements of a software system and configure runtime monitoring without bothering where the system will be deployed. At runtime, Tower 4Clouds is able to autonomously reconfigure itself when the system changes or is

migrated to another cloud provider.

In the second part, the thesis moves the focus from the challenges that monitoring tools developers have to address to the ones that monitoring tools users have to cope with. In order to understand the existing issues an empirical study was conducted in order to discover main obstacles industries are facing in monitoring their software systems. For such purpose, more than 140 practitioners from various industries were surveyed. Results showed that effective monitoring is still a difficult task, hardly affordable by small and medium enterprises with few resources and expertise. A huge number of monitoring tools, both commercial and open-source ones, proliferated in the last few years. However, no clear established solution has yet arisen. Big enterprises with high expertise such as Google, Facebook or Netflix are able to develop their own solutions to readily identify and even prevent problems or to highlight users needs, often sharing some of their tooling open-source. Enterprises with big resources and high reliability requirements, such as banks, usually pay for expensive commercial solutions. All the other companies either do not monitor, implement custom solutions or use some custom composition of monitoring tools among the hundreds of existing ones. The study, after trying to identify main challenges in the adoption of monitoring, is used to evaluate Tower 4Clouds and how it is able to address such problems.

In the third part, the thesis addressed such open challenges by proposing Omnia, an approach for structured monitoring configuration and rollout based around a monitoring factory, *i.e.*, a re-interpretation of the factory design-pattern for building and managing ad-hoc monitoring platforms. Comparing with practitioner surveys and the state of the art, Omnia shows the promise of delivering an effective solution that tackles the steep learning curve and entry costs needed to embrace cloud monitoring and monitoring-based DevOps continuous improvement.

Contents

CHAPTER 1

Introduction

In 2006, Amazon launched the first widely accessible cloud computing infrastructure service [18]: *Elastic Compute Cloud*. The need for a new server in which to deploy a new application became a simple API call. That server would cost few cents per hour until needed and the user would be charged at the end of the month on his credit card. Then, via another API call it could be shut down and the user would stop paying. Since then, software development and its release cycle in the industries have been dramatically changing. This novel compute model was soon implemented in different flavors by other big players such as Microsoft and Google. The hosting solutions proliferated and any developer with Internet access and a credit card could easily deploy large distributed software systems in minutes. In 2013, *Docker* came along [19], an open source project that automates the deployment of applications inside Linux containers by means of a simple API. Among several advantages, Docker made deployment time of a new service drop from few minutes to few seconds. In 2014 Amazon launched AWS Lambda [20], the first widely adopted implementation of the *Serverless* architecture, which introduced a new deployment model where application significantly depend on third-party services or on custom code that is run in ephemeral containers [21]. Developers deploy pieces of code which

run in few milliseconds, just the time to respond to an event, and they are charged per a hundred milliseconds.

The overviewed technological advancements are just some of the innovations that changed how software is built and released. Computing resources have been abstracted away, they became utilities, not anymore physical servers which every company had to acquire upfront. Deployments procedures which were previously performed manually are now automated so that an increasing number of industries are able to perform tens of deployments per day. In order to compete with the agility that big players are pushing forward, high levels of automation and effective monitoring of the system became some of the most important requirements for any IT department. Automation is needed to deploy new versions of a service in a safe and reproducible way. Monitoring is required to assess the quality of service and the user experience in order to understand and solve problems and take business decisions fast.

As discussed throughout this thesis, experience shows that effective monitoring is still a difficult task, hardly affordable by small and medium enterprise with few resources and expertise. A huge number of monitoring tools, both commercial and open-source ones, proliferated in the last few years, as a clear response to the need for modern solutions that can keep up with the aforementioned evolution. However, no clear established solution has yet arisen. Big corporations with high expertise such as Google, Facebook or Netflix are able to develop the appropriate solutions for their scales. Enterprises with big resources and high reliability requirements, such as banks, usually pay for expensive commercial monitoring solutions. All the other companies either do not monitor, they implement custom solutions or use some custom composition of monitoring tools among the thousands of existing ones.

In this scenario, the focus of this work is, first of all, to highlight the importance of monitoring. We try to support our claim that monitoring is a fundamental asset of the software development cycle and should be considered a first class citizen, just like automated testing, integration and deployment. We try to identify why existing monitoring tools are not able to satisfy the current technological advancements and the challenges developers of monitoring tools are facing. Then, we propose a solution able to addresses these challenges and provide future research directions on the matter. Next, we move from the challenges that have to be addressed by monitoring tools developers to the challenges that monitoring tools users have to deal with. For the purpose, we conducted an empirical study on the state of practice. We were interested in understanding how industries

are currently monitoring and handling incidents in their software systems in order to find unexplored challenges and further validate our solution, checking whether the proposed approach addresses some of the challenges users are experiencing. Finally, we prototyped a solution that tries to address the most impactful challenges, which we perceived being mainly the lack of standards, affecting the integration among developers, operators and business teams, and the growing number of monitoring tools coming out every other day. This solution offers an approach, called *Omnia*, whose key objective is reducing the learning curve and entry-cost to monitoring technology. Its main contributions consists of two major parts: (1) a common *monitoring interface* for developers and a (2) *monitoring factory* for system administrators that helps building a monitoring system from existing monitoring tools implementing such common interface.

1.1 Challenges

This thesis was developed in the context of the MODAClouds FP7 IP[1] European project. MODAClouds (MOdel-Driven Approach for design and execution of applications on multiple Clouds) had the main goal of *"providing methods, a decision support system, an open source IDE and run-time environment for the high-level design, early prototyping, semi-automatic code generation, and automatic deployment of applications on multi-Clouds with guaranteed QoS"*. Researchers contributing to these project agreed that cloud computing is a major trend in the ICT industry that can bring huge benefits especially to small and medium enterprises (SMEs) [33]. On the other hand, partners identified the lack of interoperability between clouds as a major issue that brought several open challenges such as vendor lock-in, risk management and quality assurance. Within this context, the following challenges for the design of a suitable monitoring platform were collected.

- The high market demand for cloud hosting solutions that could reduce both costs and time to market resulted in a context where each provider is offering heterogeneous solutions in terms of technologies, abstraction level, performance and costs. An increasing number of systems are developed as integrations of several systems (*System of Systems*), managed by different providers, with different priorities and release cycles. Some services may be geographically distributed on different data centers. Moreover, there is no stable hierarchy among resources

[1] `www.modaclouds.eu`

since different deployment models are used and new ones appear every day.

Requirement 1.1.1. *A modern monitoring system should be able to cope with this heterogeneity, offering a way to abstract from complexity and prevent lock-in.*

- Cloud computing brought a deployment model where resource are becoming more and more ephemeral. The things being monitored are not static servers anymore but can scale up or down in minutes, seconds or even milliseconds.

  Requirement 1.1.2. *Monitoring tools should be able to seamlessly operate on systems that are continuously evolving, with minimal or no manual intervention. Elasticity is today a first class citizen among the requirements of a monitoring system.*

- Not all cloud services offer the same flexibility. For example, services may run on platforms where agents cannot bind to addresses or cannot rely on specific libraries.

  Requirement 1.1.3. *Data collection protocols should be portable and have few requirements client side.*

- Evolution is so fast that each new technology or platform disrupt the previous under multiple aspects (*e.g.*: price, ephemerality, scale). Subsequently new vendors and monitoring tools appear in each generation trying to take over the new market share [30].

  Requirement 1.1.4. *A monitoring platform should be extensible in order to easily adapt to new requirements and to integrate with new tools and technologies.*

- New techniques, process and automation tools for increasing the speed to market are spreading across industries, especially driven by the DevOps movement. Splitting the system into microservices managed by independent teams, applying continuous delivery and feature flags, allows to push new features to production in seconds. Final users have become used to high level of performance and easily switch to competitors when availability issues are perceived. Developers should be able to check whether the new version caused problems before users can perceive it and rollback if needed [30].

Requirement 1.1.5. *Data collection should offer configurable frequency, according to requirements for each metric, and push-based so to account for events and not only on statistics.*

1.2 Contribution of the thesis

There are four major contributions in this thesis. The first two contributions address challenges relative to the current technological advancements and therefore related to monitoring tools design and development. The other two contributions are instead related to the challenges that monitoring tools users encounter and aim at understanding and helping the usability of existing solutions. This thesis main contributions are:

1. A *model-based approach* to help configuring runtime monitoring since the design time, starting from QoS requirements definition on top of a provider independent model of the system to be monitored. The configuration is based on a *rule language* which can be used to define what to monitor, how data should be filtered and aggregated, what conditions should be verified and what actions to perform (Chapter 5).

2. A *runtime multi-cloud monitoring platform* configured according to the rules defined at design time, automatically adapting to system changes and providing a portable data collection protocol able to collect data on both Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) solutions (Chapter 6). The platform is extensible and proved to be easy adaptable and integrable into different usage scenario by more than one industry (Chapter 7).

3. An *overview of the state of practice in industries*, based on in-person interviews and a survey filled in by 141 participants from the industry, which provided insight on current practices regarding monitoring and incident handling (Chapter 8).

4. The novel concept of *monitoring factory*, a re-interpretation of the factory design-pattern for building and managing ad-hoc monitoring platforms (Chapter 9).

Preliminary studies and initial implementations for the first two contributions have been published in the proceedings of the 2nd International Workshop on Ordering and Reasoning (OrdRing 2013) [44] and in the proceedings of the 3rd Workshop on the Management of resources and services

in Cloud and Sky computing (MICAS 2014) [29]. A journal paper regarding such contributions has been submitted for publication in IEEE Transactions on Software Engineering. The final implementation of the monitoring platform has been release open source on Github[2] together with a website[3] with documentation and examples and a dissemination video[4]. The third contribution has been submitted for publication in Empirical Software Engineering. The fourth has accepted for publication in the 3rd International Workshop on Quality-Aware DevOps (QUDOS 2017).

1.3 Structure of the thesis

This thesis is structured as follows:

- Chapter 2 has the main objective of a creating a common background regarding monitoring, proposing, first, a definition and reasons supporting why monitoring is fundamental in today's software release cycle as a normal prosecution of validation and verification techniques into the runtime. Then, we report the terminology and the properties that are useful throughout the thesis.

- Chapter 3 surveys whether and how the state of the art addresses the challenges identified in Section 1.1 both from the industrial and the academic point of view.

- Chapter 4 presents the Tower 4Clouds approach at high level. Then an itinerary example is described that will be used to aid the discussion throughout the thesis. We identify monitoring requirements for this example and, finally, we describe how the approach would be applied to it.

- Chapter 5 details the design phase of the approach proposed in this thesis: the modeling approach for the software system, the QoS constraints and for monitoring rules. Finally, we describe how data collectors should be configured in order to correctly work during the runtime.

- Chapter 6 describes the runtime platform, its core components, how the adaptation works, how it can be extended and how can be integrated with existing solutions.

[2] https://github.com/deib-polimi/tower4clouds
[3] http://deib-polimi.github.io/tower4clouds/
[4] https://www.youtube.com/watch?v=cizhWQpY1rk

- Chapter 7 evaluates our approach, checking how the challenges exposed in Section 1.1 are addressed.
- Chapter 8 reports the work done to understand the state of practice, that is, whether and how industries monitor their system and resolve incidents, as well as challenges they meet when using existing monitoring solutions. We report the process we faced to perform this empirical study and the results from both in-person interviews and from an online survey to which 141 practitioners accepted to respond.
- Chapter 9 describes the ongoing work regarding the *Omnia* approach.
- Chapter 10 concludes the thesis and report ongoing work on the open challenges that were highlighted during the empirical study and not yet addressed by either our approach and existing solutions.

CHAPTER 2

Background

The aim of this Chapter is to create a common background regarding monitoring. Section 2.1 focuses on finding a general definition of monitoring. Section 2.2 aims at identifying the reasons why monitoring is fundamental when running and maintaining software systems. Section 2.3 provides a list of recurring terms and their definitions. Section 2.4 defines properties of the different monitoring tools useful to provide dimensions for their classification. Finally, Section 2.5 presents a list of candidate requirements for building a monitoring platform.

2.1 What is *monitoring*?

Monitoring is a very general term whose origin comes from the Latin verb *monere* which means "to warn" or "to advise". The current definition on Oxford Dictionaries [8] is "to observe and check the progress or quality of (something) over a period of time". Restricting the domain to the engineering field, the subject being monitored is usually a system composed of components. Thus, monitoring can be redefined as "the action of observing and checking the behavior and outputs of a system and its components over time" [46]. Several other definitions of monitoring can be found, among

both academics and practitioners, when talking about monitoring software systems. Here two examples are reported. They will be discussed in order to reach a reference definition of monitoring for this thesis.

- Bertolino [27] defines monitoring as observing "the spontaneous behavior of a system" and, given a specification of desired properties, checking "that such properties hold for the given execution".
- Fatema *et al.* [35] define monitoring as a "process that fully and precisely identifies the root cause of an event by capturing the correct information at the right time and at the lowest cost in order to determine the state of a system and to surface the status in a timely and meaningful manner".

Observing a system without knowing what to check, what the correct behavior is, is not very useful. Therefore, what these definitions add is of fundamental importance, that is, verifying consistency with predefined objectives (or specifications). Moreover, monitoring, as intended in this thesis, refers to the observation of a system in its spontaneous behavior. As highlighted by Bertolino [27] this last aspect is what differentiates monitoring from testing, where instead the system is set into a synthetic environment in order to reproduce a specific behavior which is not spontaneous. The definition proposed by Fatema *et al.* certainly reports desirable properties for a monitoring system, however, since this Section is aiming at finding a general and shareable definition, optional requirements are not going to be considered. In fact, a monitoring system could be used just to check whether a requirements is satisfied without being able to provide the root cause. Correctness, timeliness and cost effectiveness, as well, are important but not mandatory. Therefore, this thesis reference definition of monitoring will be:

Definition 2.1.1. *Monitoring is the action of observing the spontaneous behavior and the outputs of a system and its components for checking consistency with a given specification.*

2.2 Why monitoring?

Every person building a new product is supposed to ask himself two questions: "am I building the product right?" and "am I building the right product?". By means of these two questions Boehm [28] informally defined software verification and validation, respectively. MonitoringSoftware verification consists in checking that the software is compliant with the spec-

ification. Software validation consists in checking that the final product fulfills its intended use.

According to the reference definition of monitoring we provided in Section 2.1, monitoring is an action that covers verification, since it aims at offering instruments to check the consistency with a specification. Most verification techniques such as theorem proving, model checking, testing and code reviews, address the design time. They all aim at universal verification. Universal verification is impossible since it is often infeasible or too expensive to predict all possible states and environments the software is going to go across. Replicating during development the same exact conditions that the system will encounter once in production is hard. Besides, software requirements change very rapidly and continuously. Thus software needs to keep adapting and evolving after the first release. Also, it is often the case that there is no single organization in control and systems have to rely on third-party services, whose QoS is seldom guaranteed and can only be monitored during the runtime. Sudden changes in third-party services QoS often require adaptation, such as scaling up or migration, which requires to be automated in order to prevent service disruption. Clearly, automation cannot work effectively without runtime monitoring information. Finally, releasing perfect software requires infinite development time. It is always a matter of tradeoffs, since unreleased software is clearly not very useful.

Monitoring enables continuous verification where design time verification is extended after deployment. It is a required technique to verify if problems arise during the runtime, after the software has been deployed. This is the only way to verify the satisfaction of requirements on traces which were not tested ahead. Once a problematic trace is detected, it can be used to create a new test, reproducing the detected issue, and avoid future regressions [45].

As experience shows [34, 43, 49] there is no single solution able to discover all bugs, multiple techniques must be used in combination in order to maximize as many faults as possible. Monitoring, should therefore be used to complement the design time verification and not to substitute it.

Finally, with the advent and large adoption of cloud computing the need for monitoring became even more important. As highlighted by Aceto *et al.* [22], this new computing model where resources can be acquired on demand and payed for the actual usage allows to reduce costs by actuating efficient capacity and resource planning and management. In order to do so, it is required to monitor costs and actual usage of resources so that automation can come into play. Moreover, performance offered by third-party services needs to be monitored in order to check if Service Level Agreements

(SLAs) are satisfied.

2.3 Monitoring terminology

This Section provides a list of definitions of terms that are going to be used throughout this thesis, with the intent of sharing a common vocabulary.

- **Metric**: a measurable property of a phenomenon that can be quantitatively determined. Example: response time is a metric measuring the "elapsed time between the end of an inquiry or demand on a computer system and the beginning of a response" [40].
- **Monitoring datum**: a single measurement of a metric. Example: the authentication service took 100 ms to respond.
- **Resource**: anything that can be monitored and, consequently, the source of a monitoring datum. Example: a web server, a database or a virtual machine.
- **Event**: any significant change in the state of a resource, which is usually identified by a monitoring datum and metadata about the context. Example: memory utilization is above the maximum threshold.
- **Action**: a response to a monitoring datum (or event) [54]. For example, an action could be the aggregation of a datum with previously received data, or the scaling up of the number of virtual machines.
- **Monitoring rule**: a recipe with instructions for the platform on what, how and when to collect and analyze data, and what actions should be taken in case specified conditions are met.
- **Monitoring tool**: a software component offering some monitoring functionality.
- **Monitoring platform**: the entire set of software components (*e.g.*, monitoring tools, queues, databases) that provide all the monitoring functionalities that are required by a company.
- **Data collector**: a monitoring tool in charge of collecting monitoring data from a resource, often called *monitoring agent*.
- **Data analyzer**: a monitoring tool that processes collected monitoring data, offering different functionalities such as filtering and aggregating.

- **Metrics observer**: whatever component or service capable of receiving and consuming monitoring data.

2.4 Monitoring dimensions

Whenever referring to monitoring and involved tools, a wide set of technologies, responsibilities, functionalities and categories is comprehended. Not all kind of monitoring tools are the same, neither they all have the same purposes. This Section defines properties and roles that are going to be used for classifying tools throughout this thesis.

A first dimension which is important to specify when classifying monitoring tools is the one based on the role they have in the monitoring activity. According to the classification introduced by Barrat [26], a monitoring tool can perform one or more of the following actions on monitoring data: (i) *collect*, (ii) *transport*, (iii) *process*, (iv) *store*, (v) *present*. What *to collect* means is defined in Section 2.3, *i.e.*, to get monitoring data from a resource. A data collecting tool is usually either a daemon running on the monitored host or an agent scraping data from monitoring APIs exposed by the resource being monitored (*e.g.*, Java Management Extension, JMX). Monitoring data can also be collected via code instrumentation, developers may use APIs to collect data. A tool *transporting* monitoring data is able to move data from a tool to another. Some tools may have transport capabilities implemented *ad hoc*, others may use existing general purpose solutions such as message brokers. Monitoring data *per se* is useless if it is only collected, it needs to be used for a purpose. Data can be *processed*, for example, to extract higher level knowledge from raw data or to verify conditions on it. Data can also be *stored*, for example in a time series database, or it can be *presented* to the user, for example via a dashboard, and let a human understand problems and patterns and take actions. Processing can be executed either in a distributed way or centrally on a single server. Distributed processing means analyzing data on several machines in parallel and eventually aggregate results on a single host. This solution can reduce network traffic and is more scalable, however it prevents more sophisticated processing algorithms which cannot be parallelized. Centralized processing requires data to be transported to a server where it is aggregated and analyzed.

Robinson [48] classifies monitoring tools based on their task, as well. In his paper he defines four different layers: the *Event* layer, which comprehend all services used for collecting, transporting, filtering and storing monitoring data or events; the *Model* layer, where events and monitoring

data are interpreted according to models; the *Presentation* layer, where services present analysis to developers; finally, the *Application* layer where application specific services, supported by the model layer, allow to present and control the software system.

Besides classifications based on roles, there are other classifications identified in literature [22] mainly focusing on how data is collected from resources. Collection can be either *passive*, also known as *non-intrusive*, or *active*, also known as *intrusive*. It is passive when there is no need to modify the resource in order to be monitored. Collecting data about the network activity through packet sniffing, for example, is considered to be passive. On the opposite, active monitoring is performed when the resource requires some modification for exposing data to the data collector (*e.g.*, API). Another dimension of classification in collection techniques is *push-mode* versus *pull-mode*. Monitoring data can either be pushed to or pulled by a monitoring tool.

Another dimension that is considered in this thesis is *stateful vs. stateless processing*. A tool, for example, can maintain information about the state of resources, a model or inter-relationships among components. On the opposite, if no information is maintained across subsequents events, the tool is said to be stateless. Stateless tools are easier to scale.

A further classification concerning how monitoring data is processed is the *expressive power* of the configuration language, which actually tells what a user can do with monitoring data. A configuration language may allow to (i) report time series with a configurable granularity, (ii) offer statistical aggregation capabilities (*e.g.*: maximum, minimum, average), (iii) define thresholds, (iv) define alarms or actions to be taken under given circumstances, (v) provide filtering capabilities.

Last, from the business perspective, a tool in general can be classified in terms of its *licensing model*, *i.e.*, open-source vs commercial, and its *deployment model*, *i.e.*, self-hosted vs cloud.

2.5 Requirements for a monitoring platform

In this section we aim at providing a comprehensive list of requirements for monitoring platforms according to a literature review [22, 35, 53] and their definitions.

Portability [35, 53] Applications can be deployed on multiple kinds of platforms. In the case of multi-cloud applications, the same service could be developed to run on totally different hosting solutions. Monitoring tools

should therefore be able to operate on heterogeneous environments.

Interoperability [35, 53] Monitoring should work across heterogeneous data centers in order to provide users the possibility to run their application on multiple clouds or hybrid solutions.

Archivability [35] Being able to access historical data and perform *post-mortem* analysis to learn from past failures and errors is an important requirement for a monitoring system.

Elasticity [22,35] Monitoring tools should continuously operate, without interruption and without any manual reconfiguration, in case of dynamic environments such as the cloud, where resources are volatile and the architecture is subject to sudden changes.

Scalability [22, 35, 53] The monitoring system should guarantee to work no matter the workload.

Affordability [35] The cost requirement is usually a trade-off which every company should face. Open-source solutions are becoming more and more popular. The growing number of contributors, even from large companies such as Google and Microsoft, allow to use high-standard software for free. However, the support is often required by most companies and they are usually willing to pay.

Non-intrusiveness [22, 35] A monitoring system, should not interfere with the normal functioning of the application being monitored, therefore it should use as little resources as possible.

Multi-tenancy [35] Cloud providers offer multi-tenant environments where multiple users usually share the same physical resources. A good monitoring platform should be able to provide visibility to both the cloud provider and its users, maintaining the respective required isolation. Cloud providers should be able to know the actual resource usage for each tenant to guarantee SLAs, while users should be able to gather information about their system and possibly an aggregated view of shared resources.

Customizability [35,53] Monitoring solution are more desirable if users have the possibility to configure runtime monitoring as they like and according to their requirements. A customizable monitoring tools should for

example offer the possibility to add custom metrics, manipulate such metrics with powerful and feature rich configuration languages, attach custom and third-party tools in a modular fashion.

Comprehensiveness [22, 53] A monitoring system should be able to monitor multiple kind of resources, both physical and virtual, multiple layers and platforms, multiple cloud providers.

Extensibility [22, 35, 53] A monitoring system is extensible if it easy to add functionalities or support for additional platforms or cloud providers, for example, via plug-ins.

Usability [35] A monitoring system is usable if it easy to install and integrate with the monitored application without interfering excessively with developers main business, offering simple, possibly self-service, insights of the application to whoever require monitoring information. Monitoring should also be easy to maintain during the evolution of the application and the changes in requirements.

Adaptability [22] A monitoring system is adaptable if it can automatically configure itself to reduce its intrusiveness at runtime, finding the right balance between monitoring requirements and invasiveness towards the running system.

Timeliness [22, 53] A timely monitoring system is able to provide the required information when it is actually needed and useful.

Autonomicity [22] Besides offering visibility of the system to a QoS engineer, an important requirement for a monitoring system is that of using monitoring data to automatically remediate system failures.

Resilience [22] A resilient monitoring system is able to continue to work or automatically recover from a complete or partial failure of some of its components.

Reliability [22] A reliant monitoring system is able to provide its service under stated conditions for a specified period of time.

Availability [22] An available system provides its functionality whenever requested by the user. A monitoring system should in general be more available than the system being monitored. Having for example the same monitoring system deployed on the same cluster of the deployed application, is a major problem in case the cluster goes down.

Accuracy [22] An accurate monitoring system is able to provide accurate metrics, which means that they are as close as possible to the real value.

Among this list of requirements identified from literature, we mainly covered the ones we described in Section 1.1, namely: interoperability, elasticity, portability, extensibility and timeliness. These requirements were chosen according to the challenges identified by partners of the MODAClouds project, whose objectives and its relation with this work are overviewed in Section 2.6.

2.6 The MODAClouds FP7 IP European project

In this section an overview of the MODAClouds FP7 IP European project main goals is given as well as an overall description of the concepts and building blocks of the MODAClouds framework. This section will help the reader to better understand the focus of the work proposed in this thesis.

The main goal of MODAClouds is to provide methods, a decision support system, an open source IDE and run-time environment for the high-level design, early prototyping, semi-automatic code generation, and automatic deployment of applications on multi-Clouds with guaranteed QoS. Model-driven development combined with novel model-driven risk analysis and quality prediction will enable developers to specify Cloud-provider independent models enriched with quality parameters, implement these, perform quality prediction, monitor applications at run-time and optimize them based on the feedback, thus filling the gap between design and run-time. Additionally, MODAClouds provides techniques for data mapping and synchronization among multiple Clouds.

All involved partners, coming from both the academia and industrial world, designed and implemented throughout a period of 3 years a set of tools (*i.e.*, the MODAClouds MultiCloud DevOps Toolbox, Figure 2.1). The main elements of such toolbox are the following:

- *Creator 4Clouds* an Integrated Development Environment (IDE) for high-level application design;

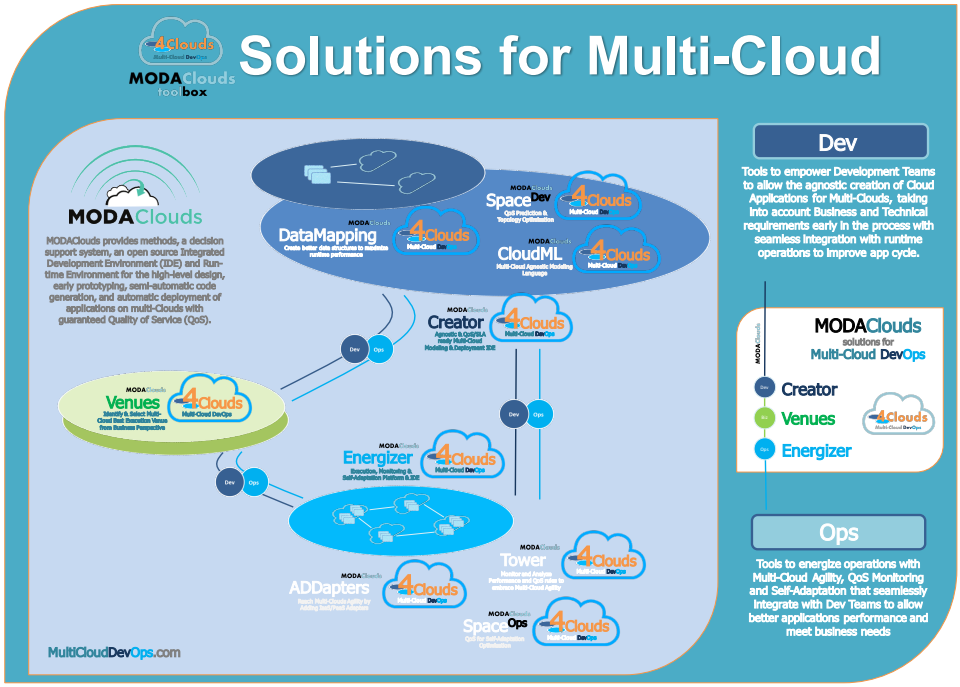

Figure 2.1: *The MODAClouds MultiCloud DevOps Toolbox*

- *Venues 4Clouds* a decision support system that helps decision makers identify and select the most suited execution environment for Cloud applications, by considering technical and business requirements;
- *Energizer 4Clouds* a Multi-Cloud run-time Environment providing automatic deployment and execution of applications with guaranteed Quality of Service (QoS) on compatible Multi-Clouds.

The Tower 4Clouds monitoring platform, which is the main contribution of this thesis, is part of Energizer 4Clouds.

The MODAClouds framework is supported since the design time by 3 main models:

- the *Cloud-enabled Computation Independent Model* (CCIM), to describe an application from the business perspective;
- the *Cloud-Provider Independent Model* (CPIM), to describe cloud concerns related to the application in a cloud agnostic way;
- the *Cloud-Provider Specific Model* (CPSM) to describe the cloud concerns needed to deploy and provision the application on a specific cloud.

Each one of these models is composed by a set of submodels that provide details about usage scenarios, service orchestration, data models, deployment models and QoS models.

The MODAClouds Model-Driven Development approach relies on the so called MODACloudsML which integrates a set of domain-specific languages. These languages cover the specification of both functional and non functional aspects of multi-cloud application. This three levels architecture provide users the ability to design multi-cloud applications in a provider-independent way.

Public deliverables of the project are available online[1] and a book was published [33].

[1] http://www.modaclouds.eu/publications/public-deliverables/

CHAPTER 3

Related Work

This chapter overviews some of the most popular monitoring tools as well as academical approaches and prototypes with the intent of checking whether and how the challenges discussed in Section 1.1 have been addressed. This Section does not aim a comprehensive survey of existing solutions. Other works [22, 35, 47, 52] already deeply analyzed and classified the state of the art of monitoring frameworks. This section only reports representative examples that helped identifying the open issues. For the purpose, part of the contribution brought by the aforementioned works will be used. Table 3.1 provide a summary of all tools and their classification based on the dimension introduced in Section 2.4.

Nagios

One of the most popular monitoring tool is Nagios [7], an open source monitoring and alerting solution released in 1999. It is an all-in-one solution, covering all roles listed in Section 2.4 (*i.e.*, collect, transport, process, store and present). Data collection is performed via scripts, or monitoring checks, periodically scheduled by the server, that provide *OK*, *WARNING* and *CRITICAL* flags. Data collection can be both active and passive:

some checks are executed on agents distributed with the system, others are probing resources remotely. Data processing is therefore mostly distributed across resources. A rich community provides several Nagios plugins open source. Transportation is implemented by using either standard protocols such as ICMP, SSH or SNMP, or their custom Nagios Remote Plugin Executor (NRPE). Local (client-side) checks results are stored in a file periodically retrieved via TCP from the server. No discovery mechanism is implemented, therefore both knowledge about the monitored system and their interdependency must be explicitly stated in the server configuration. A dashboard is provided to visualize checks results. Graphing and trending utilities can be added through plugins. Historical data is saved on rotating log files, however can be exported to external SQL data base through a plugin. Nagios Core is provided open source, however a commercial enterprise edition with additional support tools for large scale deployments is also offered.

Pros. *Offering a very simple interface for checks it is very extensible and supported by a strong open source community.*

Cons. *Since it was built before the cloud advent, it was initially developed for static servers, so it does not consider ephemerality and elasticity by design. Also, the pull based approach together with the active checks mechanism initiated by the server is not scalable when the number of resources grows.*

Sensu

Sensu [14] is an open source monitoring platform, released in 2013, sharing several concepts with Nagios approach and their check plugins, however, being a much more recent product, it was built with an eye to modern systems. Sensu aimed at offering a monitoring solution focused on the elasticity and the scalability of modern software architectures. In contrast to Nagios, which is an all-in-one solution, Sensu's main functionality is to route monitoring data. In particular, it covers the collect role by reusing the same specification of Nagios for checks, so to exploit Nagios huge community artifacts. Moreover, client side agents push data to the server (instead of being pulled by it) via a message broker (*e.g.*, RabbitMQ [11] or Kafka [1]) that completely decouples components. The state is maintained externally in a Redis [12] database so that servers are completely decoupled and easy to scale. Configuration and checks can be installed both from the server and from the client (through some configuration and management tool such as Chef [2] or Puppet [10]). Sensu clients and Sensu servers are decoupled,

the workload is distributed among clients where checks are evaluated and results are sent to the server via the queue, where multiple kind of handlers can be configured to react on events.

Pros. *It was built with cloud applications in mind, using a push based mechanism with publish subscribe patter, so it is elastic to changes and scalable. Also, it uses the same checks protocols used by Nagios, and provide plug-in architecture also for filters and handlers, therefore, extensibility is one of its strengths.*

Cons. *Agents require to bind to a port for communicating with the server, which could not be possible on some hosts, therefore it lacks in portability. It is a piece of a monitoring platform, configuration and management tools as well as other monitoring tools must be connected to create a full working platform. This flexibility can actually be considered an advantage by those who are looking for flexibility, while it may be considered a disadvantage for those who want a simple setup process.*

Collectd

Collectd [3] is a widespread monitoring tool which collects system performance statistics and its main role is that of *collecting* data. Its strength relies in performance and portability, since it is written in C and has very low CPU footprint and even with very short monitoring intervals. Community provided plugins are used to monitor multiple kind of resources. It also offers a good *transport* mechanism (Collectd binary protocol) to route data to other instances and multiple ways of *storing* data (*e.g.*, RDD files). However it provides limited *processing* functionality and it does not *present* data by means of graphs, therefore additional tools must be connected to cover the complete monitoring stack. For example it can be integrated with both Sensu and Nagios by means of the collectd-nagios check plugin.

Pros. *It is flexible, extensible and highly supported by the community.*

Cons. *Similar to Sensu, it is just a piece of a platform, plug-ins, configuration and other tools must be added to make it useful. Also, it does not offer a capability to add metadata to monitoring data by default.*

Graphite

Graphite [6] is a popular time-series database and graphing tool which offers a very simple API for pushing data to it and renders graphs of this data on demand through a customizable web based dashboard. It is composed of three software components:

- *carbon*, a daemon listening for monitoring data,
- *whisper*, the database library for storing time-series data,
- *graphite webapp*, the web application rendering graphs on-demand.

Metrics pushed to Graphite contain metadata about resources according to the schema chosen by the user. Each metric is stored in a path according to the name of the metric, each component being delimited by dots (*e.g.*, "server1.webapp1.cpu").

Pros. *It is very popular, performant and has a very powerful query language.*

Cons. *Monitoring data meta-data are expressed using the metric path protocol, which imposes a predefined schema based on the ordering. Every team could easily create its own schema and cause data misalignment. The dashboard looks quite old.*

Grafana

Grafana [5] is a popular open source graphing tool which can be used with multiple data sources. It was born and is often known as an alternative to the graphite web application, with a richer and nicer dashboard. Since version 4.0 is also offering alerting capabilities. Today it can be attached to most of the famous time series databases, such as InfluxDB, OpenTSDB or even Amazon CloudWatch. Community plugins are growing at rapid pace.

Pros. *Very nice looking interface, plug-in based and highly supported by the community.*

Cons. *It is simply a dashboard with alerting capabilities, does not provide collecting capabilities, a backend storage is required.*

Riemann

Riemann [13] is an event-based open source tool for monitoring distributed system. It has a very high throughput (a million of events per second at sub-millisecond latencies on commodity hardware), it is highly configurable and written in Clojure, which is also the base language for the DSL used for configuration. Events are pushed to Riemann, processed and then exported to other systems, very similarly to a router. Each received event is added to one or more streams. Indexes are maps used to maintain the last received event for each host/service pair until it is invalidated by a timeout (TTL).

Pros. *It is flexible, highly scalable and extensible.*

Cons. *It is not an all-in-one monitoring solution, it can be used as a router or data analyzer. Configuration is done via Clojure code, which make it flexible but requires upfront investment in learning and writing the configuration.*

StatsD

StatsD [15] is an open source daemon, developed by Etsy, listening for monitoring data and sending aggregates to one or more pluggable backends (*e.g.*, Graphite). The main strengths of StatsD are:

- simplicity, it is very simple to instrument code,
- ubiquity, there are client libraries for multiple languages,
- small footprint, clients have negligible overhead.

Pros. *Very simple and compact protocol.*

Cons. *It is just a collector and aggregation point. It support metrics according to graphite metrics path schema, without tags support by default.*

Prometheus

Prometheus [9] is an all-in-one monitoring platform, open sourced in 2015. Even though it is very recent, it goes against the current monitoring tools' trend of using a push based mechanism. Instead, it resemble Nagios in how monitoring information is retrieved from resources being monitored. The main difference with the data collection activity is that instead of using check scripts, Prometheus only collects time series data from a set of instrumented targets over the network. Moreover, all metrics are collected via HTTP, in a parallel and more efficient way than Nagios.

Pros. *There is no central server, every team can deploy its instance and start monitoring instrumented target. It is an all-in-one monitoring tool with its own dashboard, storage and configuration language.*

Cons. *It must rely on resource discovery tools and it does not deals with events but only with aggregations. Meta-data regarding the resources being monitored is not univocally defined, it lacks a central source of knowledge and it is still entirely conveyed with the datum. Every team could easily create its own tags and cause data misalignment.*

JCatascopia

JCatascopia [51] is a cloud monitoring platform, which copes with the volatility of resources by implementing a self-registration protocol together with a heart-beat mechanism that allow automatic agents discovery and removal. It also features an adaptive filtering capability which avoids to send monitoring data if there is little variance with the previous value sent. This allows to reduces costs and overhead. The configuration language is based on the *Metric Subscription Rule Language*, a DSL they defined to instruct the platform what to collect, how to filter metrics and aggregate them, and what actions to perform according to conditions.

Pros. *It is elastic and scalable platform with a powerful and simple configuration language.*

Cons. *The configuration language explicitly targets resources by ID, therefore the language does not provide the flexibility required to address machines by class or resources. Moreover, agents require to bind to a port for communicating with the server, which could not be possible on some hosts.*

Dautov *et al.*

Dautov *et al.* [31] proposed an autonomous framework where monitoring data is annotated with semantic information using the Resource Description Framework (RDF) language specification and semantic query languages, such as C-SPARQL [24] and SWRL [16], are used to implement reasoning against streaming information.

Pros. *This approach is valuable since it enables semantic inference and reasoning on monitoring data, based on a background ontology. A background ontology can store inference rules to define the subclass or the containment relationships.*

Cons. *The usage of a general purpose language such as C-SPARQL requires high level of expertise. The expressiveness of the underlying query language and the ability to reason on both static and stream knowledge affects considerably the scalability. There are however ongoing studies on how to parallelize the computation to increase the scalability [41].*

The Multi-layer Collection and Constraint Language (mlCCL)

The Multi-layer Collection and Constraint Language (mlCCL) [25] is an extensible language for defining how to collect, aggregate, and analyze runtime data in a multi-layered system. ECoWare is an event correlation and

aggregation framework that supports mlCCL. Data in mlCCL is described as Service Data Objects (SDOs), which are timestamped and exchanged via an event bus. SDOs also carry an instanceID, that is a unique ID identifying the specific service call. The approach distinguishes between two different sources: messages refer to the request or response messages that are exchanged during service invocations and indicators represent information obtained periodically about a service.

Pros. *The proposed framework and language address heterogeneity by providing multi-layered indicators: Key Performance Indicators for software services metrics and Resource Indicators for infrastructure service metrics.*

Cons. *The approach does not consider ephemerality of modern cloud solutions since resources are addressed by ID.*

ReMinds

REquirements Monitoring INfrastructure for Diagnosing Systems of Systems (ReMinds) [53] is a framework for developing monitoring solutions for systems of systems (SoS). Its main focus is industrial automation, where multiple heterogeneous systems require to be monitored. ReMinds proposes the definition of domain specific event based models for the integration of multiple probes collecting events from systems written in different languages and at different level of abstractions. The proposed language allows to predicate on top of such high level event model and monitor the correctness of the SoS system.

Pros. *ReMinds addresses the problem of monitoring heterogeneous systems by using a unified event model for integrating events and predicate at high level of abstraction.*

Cons. *The approach does not addresses the ephemerality of modern cloud systems.*

Tool	Collect	Transport	Process	Store	Present
Nagios	Active / Passive, Pull	NRPE / ICMP / SSH / SNMP	Decentralized, Stateless	Log files / SQL (plugin)	Dashboard / Graphing (plugin)
Sensu	Active, Push	RabbitMQ / Kafka	Decentralized, Stateless	Redis	N.A.
Collectd	Active, Push	Binary protocol	N.A.	RDD files	N.A.
Graphite	N.A.	N.A.	Centralized, API for filtering and aggregating	Whisper	Graphite Webapp
Grafana	N.A.	N.A.	Centralized, statistics	N.A.	Grafana Webapp
Riemann	N.A.	TCP, UDP, Websockets	Centralized, Stateful	Most recent information is maintained in memory	N.A.
Statsd	N.A.	UDP	Centralized, Statistics	N.A.	N.A.
Prometheus	Active / Passive, Pull	HTTP	Centralized, SQL like language for filtering, aggregating, alerting	Level DB for indexes, custom storage layer for sample data	N.A.
JCatascopia	Active, Push	HTTP, pub/sub	Centralized, Custom DSL (Metric Subscription Rule Language) for filtering, aggregating and configuring actions	MySQL	N.A.
Dautov *et al.*	N.A.	HTTP	Centralized, Stateful, C-SPARQL and SWRL semantic languages for performing inference, filtering, aggregating	RDF triple store	N.A.
mlCCL	Active, Push	Publish/Subscribe Event Bus	Centralized, Stateless, custom DSL for collecting, aggregating and analyzing	persistent storage with 24h rotation	Custom dashboard for event correlation
ReMinds	Active, Push	Socket, JMS, RMI	Centralized, Stateless, DSL for crosscutting constraints validation	Oracle / Postgres / MySQL, MongoDB, HDFS	Custom dashboards for managing constraints, reviewing events and violations, visualizing runtime events

Table 3.1: *Tools classification*

CHAPTER 4

The Tower 4Clouds Approach

In this Chapter we aim at providing an overall description of the core contribution of this thesis: the Tower 4Clouds approach for designing and monitoring multi-cloud applications.

In Section 4.1 we provide a high level overview of the Tower 4Clouds approach and show how it integrates within the entire MODAClouds framework. Next, we will present Ticket Monster, a Java EE demo application developed by JBoss[1], and its candidate monitoring requirements. Such application will be used as main itinerary example throughout the thesis to simplify the discussion and provide practical usage scenarios on how to monitor an application that can be deployed on multiple clouds (Section 4.2). Finally, Section 4.3 will show how the proposed approach can be applied to our itinerary example, from the design through the runtime.

4.1 Overview of the approach

Our approach enforces developers to consider monitoring a first-class citizen in the application development cycle. Quality of Service (QoS) should be considered since the first phases, when requirements are elicited and

[1] http://www.jboss.org/ticket-monster/

defined in a specification. Once the application specification is defined is translated into an abstraction of the real application: a *model*. According to our approach, QoS requirements (or QoS constraints) should be modeled together with the application by annotating the modeled components. These annotations would be the starting point for configuring the monitoring activity and guarantee the satisfaction of non-functional requirements during the runtime. Tower 4Clouds only addresses QoS requirements that can be quantitatively determined. Some non-functional requirements (*e.g.*, usability, security or compliance) cannot be thoroughly described by means numerical values and are out of the scope of this work.

The Tower 4Clouds approach is based on the following concepts:

- the *base meta model*, which is a set of classes and relationships among these classes that are common to all software systems and that can be monitored;
- the application specific *meta model*, which is an extension of the base meta model with classes relative to the domain of the application being designed;
- the *deployment model*, which is an instance of the application specific model, it describes a deployment scenario of the application being modeled and has a one to one mapping with MODAClouds models (Section 2.6);
- *QoS constraints*, which are annotations on top of the deployment model, specified by the QoS engineer, describing the allowed values for a given QoS metric;
- *monitoring rules*, which are annotations on top of the deployment model, used by the QoS engineer to configure the runtime monitoring in order to guarantee the satisfaction of QoS constraints;
- the *runtime model*, which is an instance of the deployment model, representing a running application, automatically kept synchronized with the actual deployment by the Tower 4Clouds framework;

Modeling QoS constraints is the first step of our approach. Within the MODAClouds framework, such constraints are annotated on top of components modeled in the cloud-independent models (*i.e.*, CCIM and CPIM, described in Section2.6). Components that can be annotated are named, according to our reference terminology (Section 2.3), *resources*, which represent things that can be monitored. Examples of resources are a service or a virtual machine.

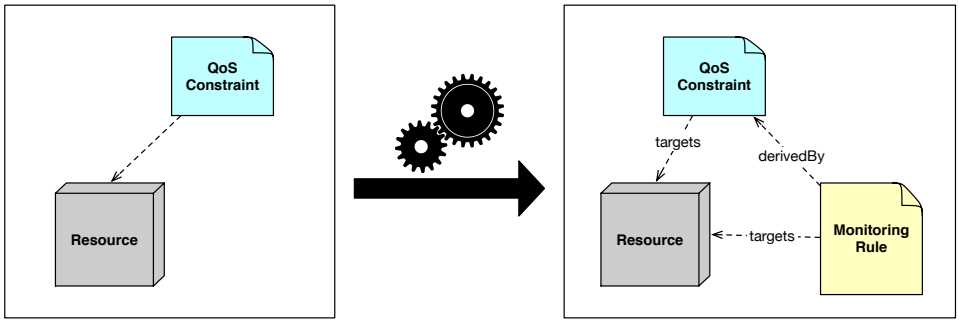

Figure 4.1: *Monitoring rule generation from a QoS constraint*

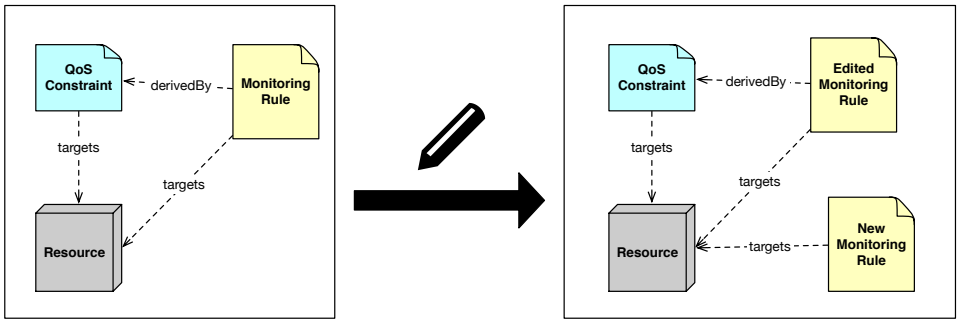

Figure 4.2: *Monitoring rule edit and creation*

Once QoS constraints are modeled, *monitoring rules* for checking the satisfaction of such constraints can be automatically generated via a model to model transformation we specified. This transformation was also implemented in form of a Java library that can be integrated in any Java modeling framework (Figure 4.1). Monitoring rules are recipes that are used to configure runtime monitoring. They allow to instruct the platform targeting resources at different levels of abstractions. A rule can, for example, target a whole class of resources (*e.g.*, all virtual machines) or resources belonging to a type of resources (*e.g.*, all web servers). Monitoring rules that are automatically generated from QoS constraints are created in order to have the platform check and alert when the respective QoS constraints are violated. Generated rules have default configurations which can be edited by the user. Moreover, the user can add new ones in order to customize runtime monitoring (Figure 4.2). For example, he can change how often the rule should be evaluated or add additional actions to be executed when the QoS is violated. We will provide a detailed specification of the monitoring rules language in Chapter 5.

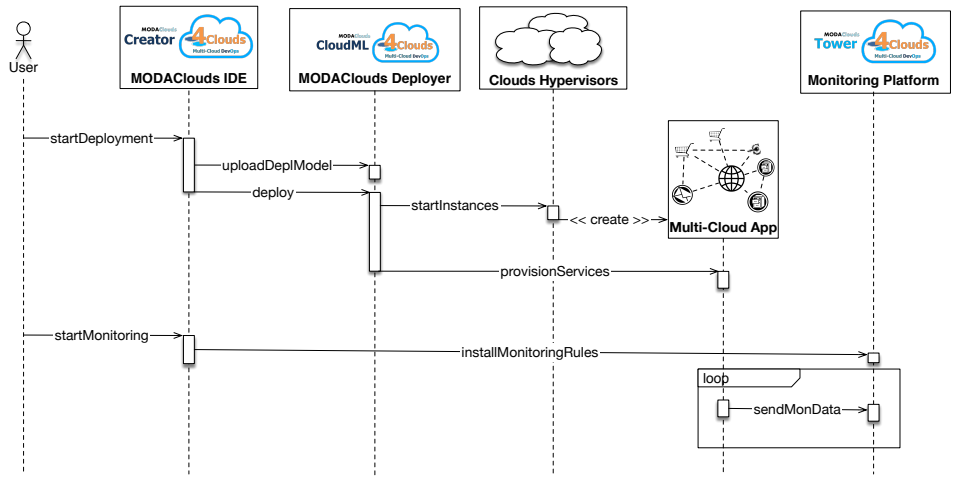

Figure 4.3: *Sequence diagram of the multi-cloud application deployment and monitoring rules installation*

Besides monitoring rules, in order for our runtime monitoring platform to be able to monitor resources, *data collectors* must be deployed together with the application. According to our reference terminology (Section 2.3), data collectors are software components in charge of collecting monitoring data from a resource. They should be modeled, and possibly monitored, as any other resource belonging to the application.

Within the MODAClouds framework, the entire design process has been integrated in the MODAClouds IDE *Creator 4Clouds* [36]. Once a first release of the application is completely modeled and binaries have been released on an artifact repository, the user can deploy the application and the monitoring rules. Both actions are triggered within the IDE. The deployment of the multi-cloud application is managed by the MODAClouds deploying service *CloudML 4Clouds* [37]. Both *CloudML 4Clouds* and Tower 4Clouds are supposed to be already up and running. A sequence diagram describing at very high level how this integrated process works is depicted in Figure 4.3.

Figure 4.4 depicts the Tower 4Clouds runtime architecture. The *Manager* is the component in charge of the management of the platform. It is responsible of configuring other components according to monitoring rules and managing the resources model. It offers both a REST API for tools integrations (*e.g.*, with an IDE) and a Web application for manual management. The *Data Analyzer*, as defined in Section 2.3, is the component responsible of processing monitoring data, filtering and aggregating it, and

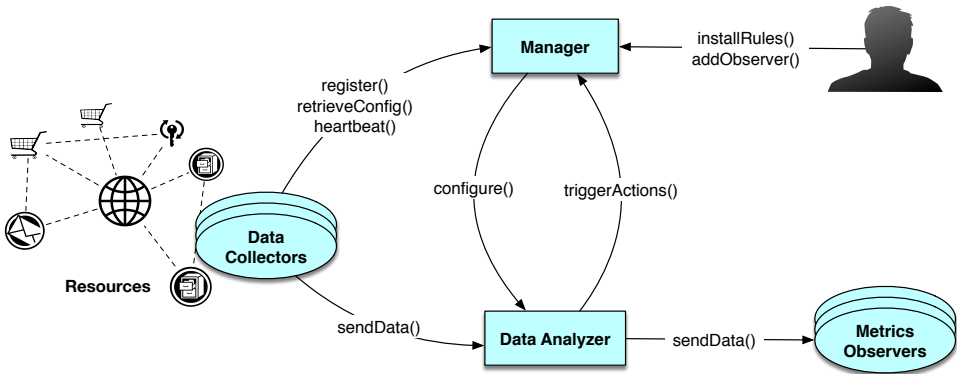

Figure 4.4: *Tower 4Clouds runtime platform architecture*

sending results to metrics observers. A *Metrics Observer* is any component that can receive metrics from the Data Analyzer. Finally, *Data Collectors* are components that are distributed on the resources being monitored.

The Tower 4Clouds runtime platform is able to autonomously react to changes in the system without interrupting monitoring. Whenever new resources are started, the existing configuration described by the monitoring rules will be applied also to the new entry.

The platform was designed so to be easily extensible in order to be able to interoperate with existing monitoring and alerting tools. Support for some of the most common protocols (*e.g.*, Graphite, Influxdb) were added by simply implementing the provided serialization interfaces. Support for some of the most common collecting tools (*e.g.*, Collectl and Sigar) were added by wrapping them using our data collector library.

The entire design and deployment phase can be continuously iterated. As the application evolves and new features are developed, the QoS Engineer is required to maintain QoS constraints and monitoring rules aligned with model changes. A modeling IDE, such as the one provided by the MODAClouds framework, can help identify issues and inconsistencies in the model. Once a new release is ready to be deployed, the set of rules will be updated as well and monitoring will be setup for the new application version.

4.2 Ticket Monster: an itinerary example

When looking for an example application that could guide the discussion throughout the thesis we aimed at finding an existing solution that could be deployed on multiple clouds, both on PaaS and IaaS, in order to be able

to describe how our approach can help addressing not only multi-cloud hosting, but also heterogeneous deployments. The application had also to be open source, so that we could be able to instrument it for collecting application level metrics.

We selected *Ticket Monster*[2], a Java EE 6 demo application developed by JBoss with the purpose of demonstrating how to build modern web applications. The application is an online ticketing broker. It met our requirements since it is open source[3] and because a Java EE application can be deployed both on an application server within any virtual machine and on PaaS such as OpenShift[4]. Moreover, it is a moderately complex application, and not just a toy application.

The application provides both an end user and an administrator interface. End users can:

- look for current events;
- look for venues;
- select shows (events taking place at specific venues) and choose a performance time;
- book tickets;
- view current bookings;
- cancel bookings.

We reported in Figure 4.5 the end user use case diagram published on JBoss website. Administrators can:

- add, remove and update events;
- add, remove and update venues (including venue layouts);
- add, remove and update shows and performances;
- monitor ticket sales for current shows.

We reported in Figure 4.6 the administrator use case diagram published on JBoss website.

The application was developed using Java EE 6 services to provide business and persistence functionalities, such as CDI, EJB, JPA. JPA specification allows to connect to several relational databases, such as PostrgreSQL

[2] http://www.jboss.org/ticket-monster/whatisticketmonster/
[3] https://github.com/jboss-developer/ticket-monster
[4] https://www.openshift.com

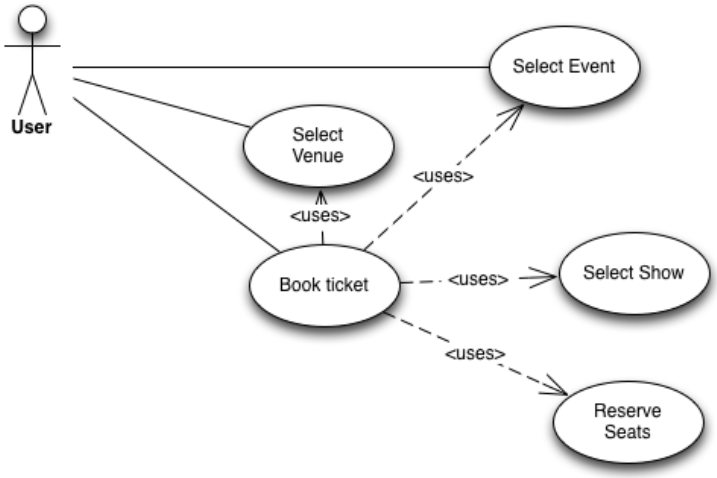

Figure 4.5: *Ticket Monster end user use case*

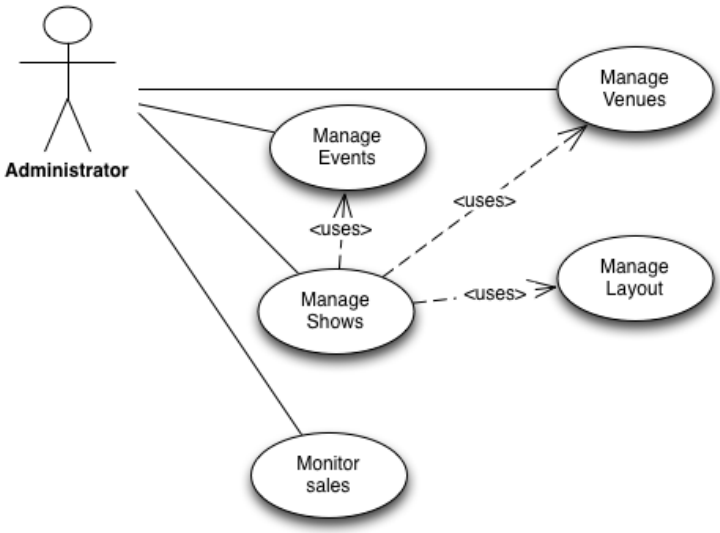

Figure 4.6: *Ticket Monster administrator use case*

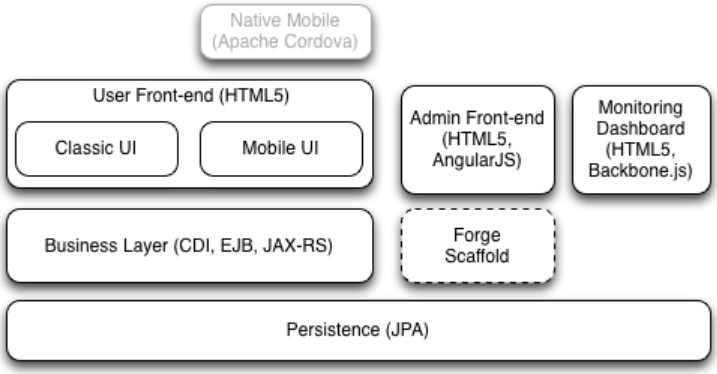

Figure 4.7: *Ticket Monster architecture*

or MySQL. JBoss developed four different UIs: two for the end-user (a web app and a mobile app), one for administration and one for monitoring sales. The architecture published on JBoss website is reported in Figure 4.7.

4.3 Our approach in action

A booking application such as Ticket Monster, is often subject to spiky traffic loads. In fact, tickets for events are usually sold in limited quantities. Therefore, as soon as new ones become available, there is often a huge number of requests coming to the server. Monitoring is essential to have a clear feedback on users experience, high latencies can ultimately result in loss of money.

In this section we aim at describing the Tower 4Clouds approach, applied to the Ticket Monster itinerary example we described in Section 4.2. The objective is to highlight the contribution from the user perspective, postponing to the next chapters the details of our solution.

4.3.1 Provider independent multi-cloud modeling

A provider independent multi-cloud model of the application is required in order to correctly design QoS constraints and monitoring rules according to the Tower 4Clouds approach. We started modeling Ticket Monster according to the IaaS deployment model (Figure 4.8). We represent resources managed by the provider with gray boxes. In the case of IaaS, virtual machines are the managed resources. We modeled two types of virtual machines, one hosting the database and one hosting the core application.

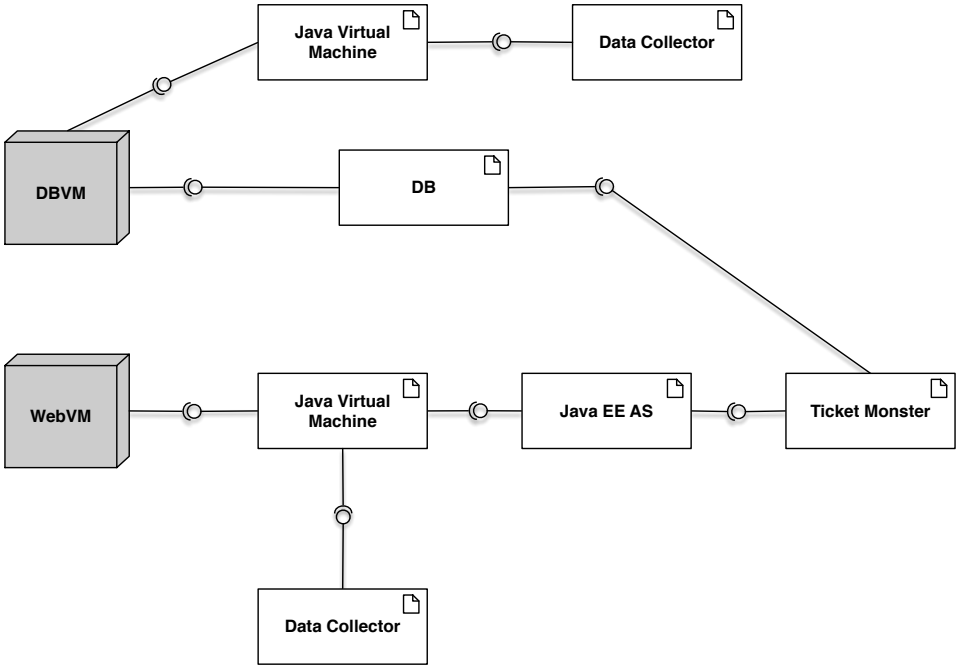

Figure 4.8: *Ticket Monster IaaS deployment model*

Ticket Monster requires a Java EE application server. Both the data collector developed for the MODAClouds project and the Java EE application server requires the Java virtual machine to run.

As anticipated, the Ticket Monster application can also run on a Java EE compatible PaaS solution. A PaaS can usually offer both the database and the application container. However, for the demonstration purposes of this work, we designed the entire system to have a common shared database on a IaaS provider and have only the Java application deployable on both PaaS and IaaS. The deployment model with both PaaS and IaaS hosting is represented in Figure 4.9.

We now have a model of an application that can be deployed migrated from one cloud to another or even simultaneously run multiple clouds.

4.3.2 Modeling QoS constraints and monitoring rules

We can now add a QoS constraint on the response time. The Ticket Monster application is annotated with a constraint stating that the 95th percentile of the response time should never exceed 5 seconds (Figure 4.10) The required rule is automatically generated as shown in the figure. The generated rule is automatically performing the action of producing a new metric named

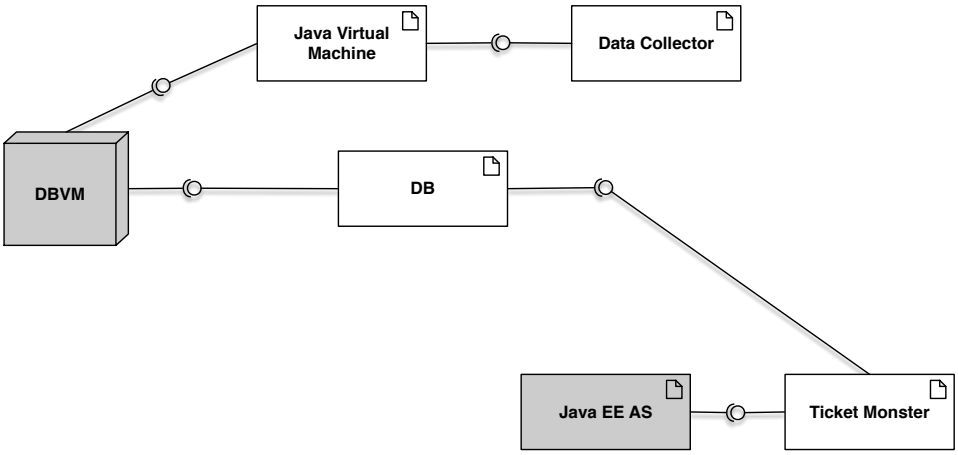

Figure 4.9: *Ticket Monster mixed IaaS and PaaS deployment model*

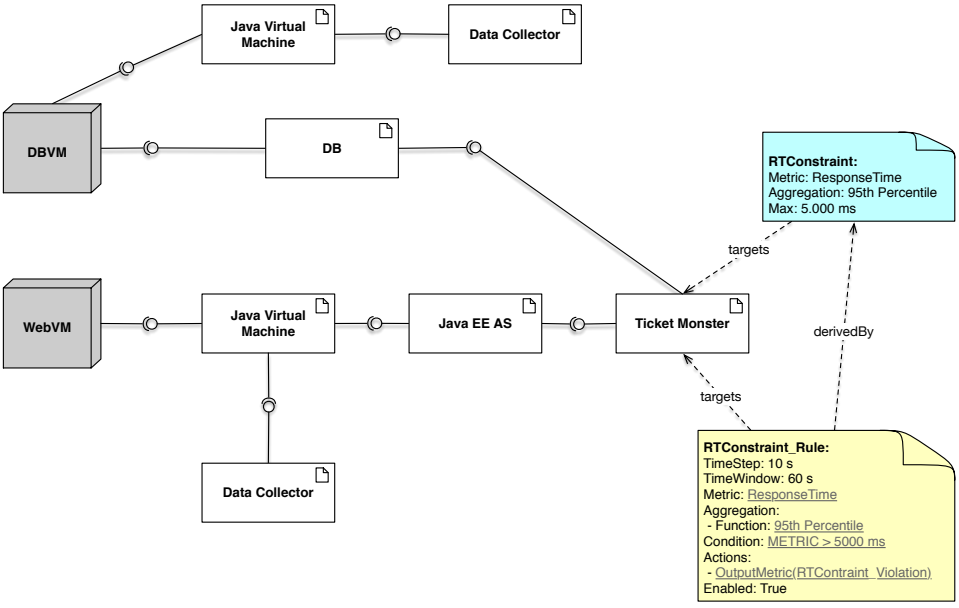

Figure 4.10: *Ticket Monster with QoS constraint on the response time*

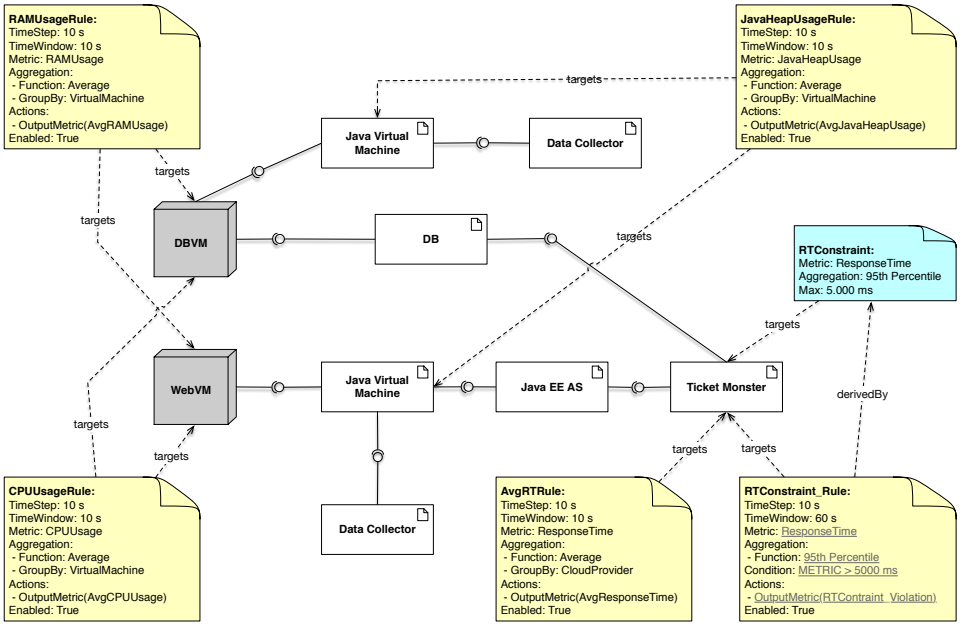

Figure 4.11: *Ticket Monster with additional monitoring rules*

RTConstraint_Violation which can be observed by any metric observer. A metric observer could be a dashboard, which switches a red light on when such violation is received, or else a mail notification service. The rule can be changed on all of its fields except for the ones inherited from the QoS constraint (underlined and grayed out in the figure). For example time steps and time windows can be changed in order to have either a finer or coarser granularity.

A notification is useful, however the user should also have some context to diagnose the root cause of the identified violation. For this purpose, some other metrics that the user can correlate with the raise of such event can be monitored. For example, in Figure 4.11 we added some manual monitoring rules to instruct the monitoring platform to monitor the CPU and RAM usage of virtual machines as well as a rule for monitoring the Java heap memory usage. These metrics can then be plotted on a dashboard.

Going back to the PaaS version of the model, system metrics such as CPU and RAM usage are usually not accessible, therefore monitoring rules for these metrics will not have any effect on such deployment. However, application level metrics remain the same and there is nothing else to change. Figure 4.12 shows the part of the mixed PaaS and IaaS model with required monitoring rules for monitoring the response time.

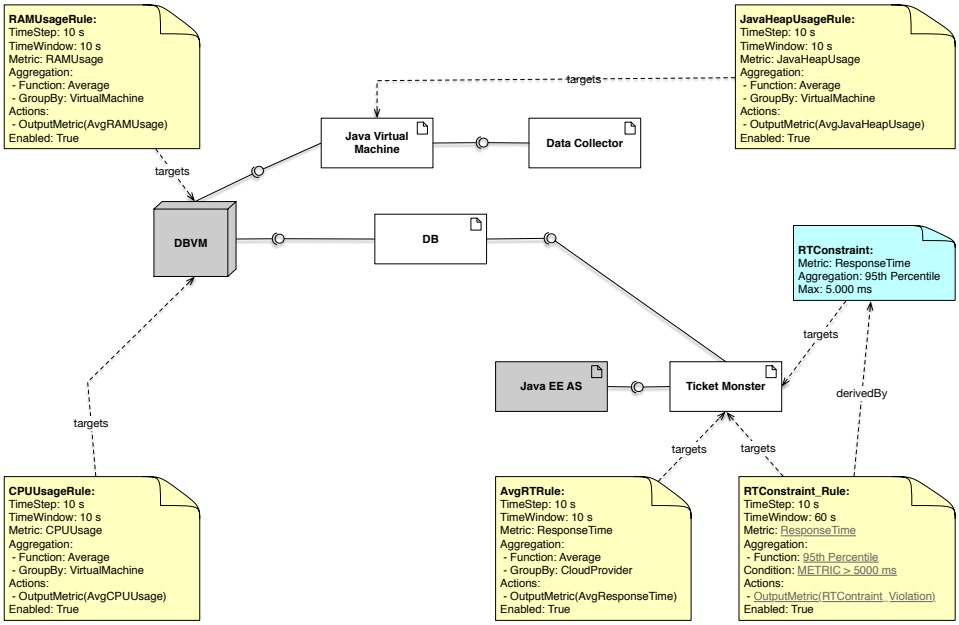

Figure 4.12: *Ticket Monster with monitoring rules on the mixed PaaS and IaaS model*

4.3.3 System and application level data collection

In the previous section the Ticket Monster example was modeled together with monitoring information. Among the software components, a *Data Collector* was modeled. The data collector we referred is a daemon that was developed together with Imperial Collage London, integrating existing monitoring tools, such as Sigar and Collectl, for their usage within the Tower 4Clouds framework (see Section 6.4.1). CPU, RAM and JVM metrics can be collected using this data collector. In order to collect application level metrics, however, this data collector cannot be used since the code requires to be instrumented.

In order to instrument Java applications we developed a library which exploits reflections and aspect oriented programming to lighten developers effort. In this section we provide a description of how we enabled response time monitoring in Ticket Monster.

The Java data collector is able to monitor Java EE applications by simply weaving in code at runtime before and after the execution of methods with JAX-RS annotations (*i.e.*, *GET*, *POST*, *PUT* and *DELETE* endpoints) and measure the duration. The developer is only required to add a few lines of code to be executed when the application is launched, in order to configure the data collector with information regarding the monitored resource and

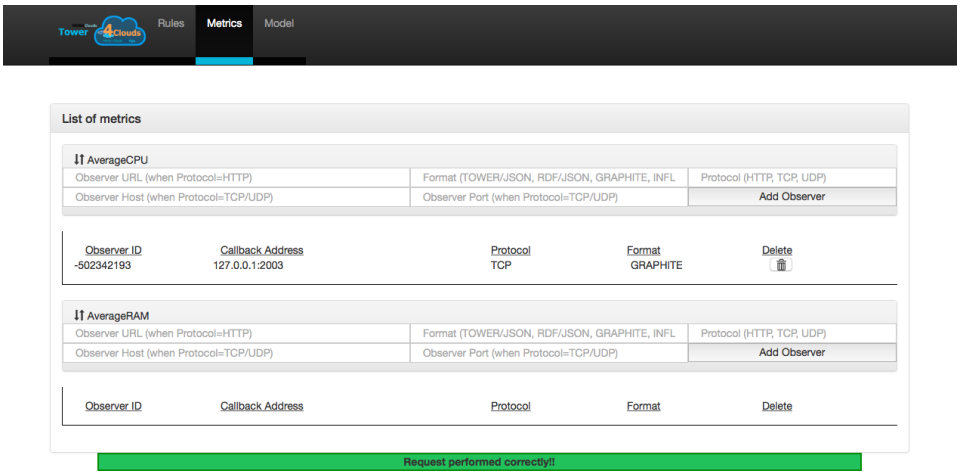

Figure 4.13: *Metrics tab on the Tower 4Clouds Manager administration page*

the *Tower 4Clouds Manager* endpoint. This information can be simply provided via environment variables. Finally, the package containing the classes to be monitored must be specified.

The source code of the class that was added to Ticket Monster in order to instrument it with our Java Data Collector library is available in Appendix A.

4.3.4 Elastic runtime monitoring

Once the multi-cloud provider independent model and monitoring rules are designed and code is instrumented, the application is ready to be deployed and monitored. As soon as the application and the data collectors are started they will contact the Tower 4Clouds manager and update the central model with the existing resources. Periodically, they will reconnect to the server to notify their liveness and retrieve updated configuration. Once monitoring rules are installed, data collectors will start sending data as specified. By means of the Manager administration web page the user can review monitoring rules, exposed metrics and the runtime model of the system. Under the *Metrics* tab of the administration web page (Figure 4.13) a user can attach metrics observers to route metrics generated by the monitoring rules to other monitoring tools. For example we attached Graphite as observer of the average RAM usage metrics and have them plotted on Grafana

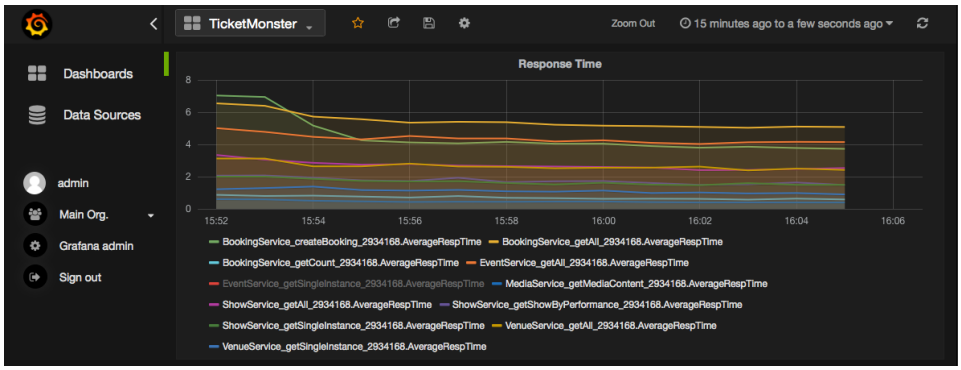

Figure 4.14: *Plotting metrics to Grafana*

(Figure 4.14). Our monitoring platform is able to elastically adapt to deployment changes such as scaling activity or migrations, without any need to either restart it or modify the configuration. In the following two chapters we will report implementation details of our Tower 4Clouds solution.

CHAPTER 5

Modeling with Quality in Mind

This Section provides details regarding the Tower 4Clouds approach as far as the modeling phase is concerned. The model defined during the design phase allows the runtime platform to be aware of the semantics behind the monitored resources. The meta-model we specified in Section 5.1 describes classes and relationships among resources and is required by the user to correctly model the system, QoS constraints and monitoring rules. In Section 5.2, we define QoS constraints and how they can be modeled. In Section 5.3 we provide the specification of the monitoring rules language, how rules can be obtained from QoS constraints and how runtime monitoring can be modeled. Finally, we describe how to model and configure data collectors (Section 5.4).

5.1 The base meta-model

The base meta-model is a set of classes and relationships among these classes that describes common concepts among all software systems. The base meta-model proposed by Tower 4Clouds approach is depicted in Figure 5.1. First, the class *Resource* is defined as the parent class describing anything that can be monitored. All other classes available in the meta-

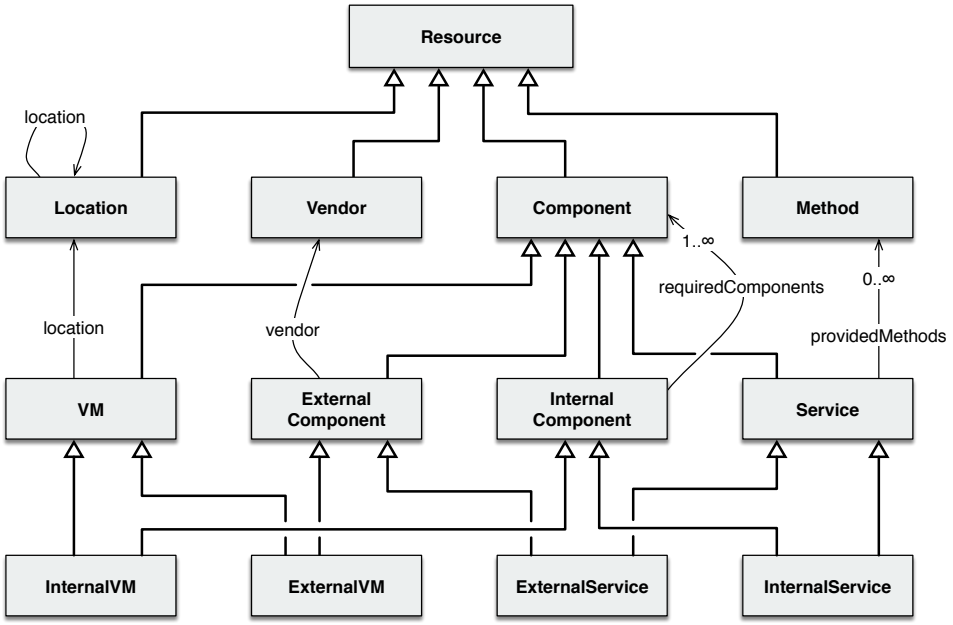

Figure 5.1: *The Tower 4Clouds base meta-model*

model are subclasses of the Resource class. The base meta-model defines the following additional classes:

- *Component*, a general class representing any subsystem. Each component can be either a *VM* (*i.e.*, a virtual machine) or a *Service* (*i.e.*, a running software application). Each component can also be either provided by a third-party vendor (*ExternalComponent*) or managed by the application provider (*InternalComponent*). Moreover, both services and virtual machines can be either external or internal components.

- *Method*, a class representing operations, or functions. A service can provide multiple methods.

- *Vendor*, a class representing a third party provider. This is also a resource since an aggregated monitoring datum can refer to a vendor (*e.g.*, the average CPU on a single cloud provider).

- *Location*, a class representing a physical or virtual area where virtual machines are located. This is also a resource since an aggregated monitoring datum can refer to a location (*e.g.*, the average CPU on a single location).

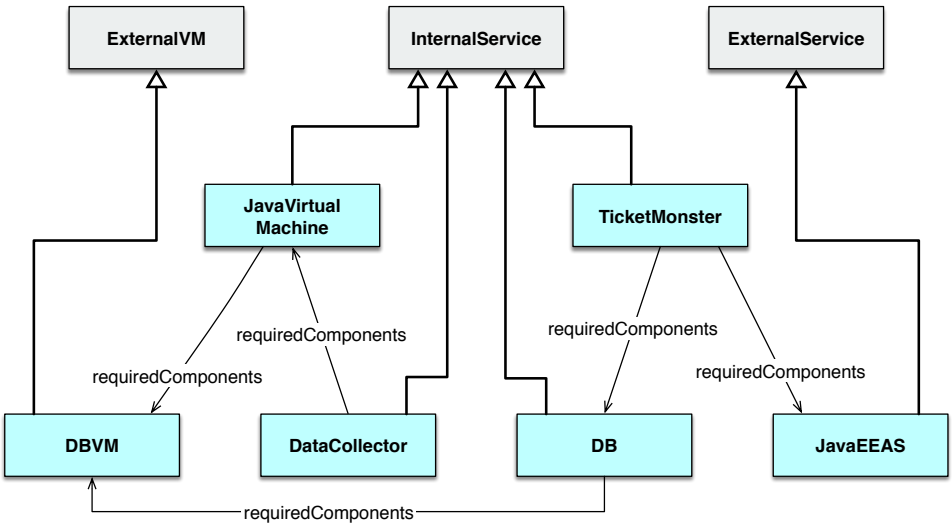

Figure 5.2: *Base meta-model extension derived from Ticket Monster model*

When a Tower 4Clouds user is designing his application and deployment model, he is actually extending this application independent meta-model with application specific classes and relationships. If we took, for example, the mixed IaaS and PaaS version of the deployment model of Ticket Monster, depicted in Figure 4.9, the underlying extension would be as shown in Figure 5.2.

Within the MODAClouds project, the UML specification was used as base modeling language, given its popularity. However, UML deployment model diagram does not have any concept of inheritance, therefore the modeling capabilities offered by the Tower 4Clouds approach are limited when using such specification and new classes can only be direct children of the base meta-model classes. However, our approach allows to define more complex hierarchies which will offer much more flexibility when defining monitoring rules. For example, the model depicted in Figure 5.2 can be further extended by creating a class *JavaApplication* as common superclass of both the *TicketMonster* and the *DataCollector* internal services.

We will see in the next sections how this is an advantage when modeling QoS constraints and monitoring rules.

5.2 QoS constraints specification

During the design phase, users can annotate resources with QoS constraints. QoS constraints are defined by the *target resource*, the *metric name* to

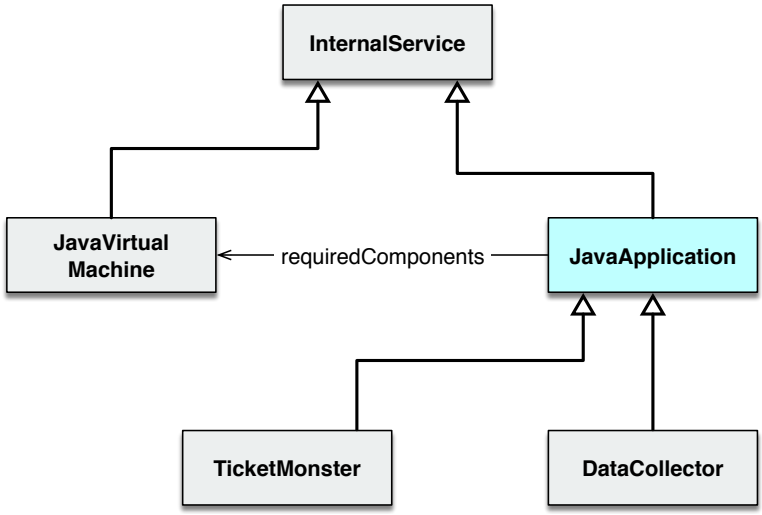

Figure 5.3: *Example of model which cannot be represented in UML*

which the constraint is enforced and the set (or range) of *acceptable values*.

The *target resource* is any class within the ones specified in the application specific meta-model. As we said in Section 5.1, in the UML deployment model diagram there is no concept of inheritance, therefore annotating the resources will limit QoS constraints to target only the direct children to the base meta-model classes. However, the Tower 4Clouds approach does not enforce this limitation and the target can be any class within the model. For example, a QoS constraint could target the InternalVM class, so that the constraint will be applied to all its subclasses. Also, going back to the example in Figure 5.3 a QoS constraint could target the *JavaApplication* class so that the constraint can be specified once for every Java application.

The *metric name* is a string which identifies a metric. Such field can be any name among the ones provided by the data collectors that will be used during the runtime. A list of provided data collectors and available metrics is reported in Section 6.4.1.

Acceptable values is a set of number whose units is dependent on the specific metric and data collector implementation. It is usually expressed by expressing a maximum or minimum value.

Within the MODAClouds project context, the same QoS constraints defined here are also used at design time by the Creator 4Clouds module Space4Clouds [23] in order to derive a deployment configuration capable of minimizing the cost of the cloud infrastructure and maintain the desire QoS.

Via the model-to-model transformation defined in Section 5.3.1, the information provided by QoS constraints is used to automatically generate monitoring rules specifying how the runtime platform will check whether these constraints are violated.

5.3 The monitoring rules language

Monitoring rules allow to define since design time how Tower 4Clouds should behave at runtime, by directly annotating the modeled resources. The language aims at enabling users to easily predicate over a multi-cloud environment at different levels of granularities. Rules instruct how and what resources should be monitored, whether any aggregation should be performed, whether any condition should be verified and what actions should be performed.

Monitoring rules are automatically generated from QoS constraints. Since rules are used not only to check constraints but also, for example, to aggregate and generate new metrics to be sent to observers, the user can also create rules from scratch that are not related to a QoS constraint.

Like QoS constraints, monitoring rules target resources by class, at different levels of abstractions. The same discourse highlighted in Section 5.2 is still valid for rules, that is, the UML diagrams are limited in comparison to the capabilities offered by the Tower 4Clouds approach. Monitoring rules, as well as QoS Constraints, can target any class of the application specific model obtained by extending the base meta-model, as described in Section 5.1.

In addition to what QoS constraints provide, monitoring rules also offer the ability to target resources by ID. IDs are only known at runtime and is the unique name of an instance of a resource. This usage is useful only at runtime, when resources are already instantiated and IDs are known. This last feature clearly limits the flexibility of the approach, however we think it may be required in some cases and we provided this functionality.

When targeting a resource by class, all resources of the instances of the class are monitored according to the rule. Multiple targets can be listed in the same rule.

Besides the targets, the monitoring rules language provide the following fields:

- *id*, the unique identifier of a rule.
- *timeStep*, the interval in seconds between two following evaluation of the rule.

Table 5.1: *Available aggregate functions*

Aggregate Function Name	Description	Required Parameters
Average	Average value	none
Max	Maximum value	none
Min	Minimum value	none
Count	Number of monitoring data	none
Sum	Sum of all values	none
Percentile	Percentile of the order given by the parameter	*thPercentile*: integer in [0,100]

- *timeWindow*, time range in seconds in which monitoring data should be considered at each evaluation step.
- *enabled*, specifies whether the the platform should start evaluating the rule as soon as the rule is installed (rules can be enabled and disabled at runtime).
- *metric*, the name of the metric to be collected together with any parameter required to configure data collection.
- *aggregation*, an optional field, it specifies the name of the aggregate function to be computed on the windowed monitoring data at each evaluation. This aggregation will be executed server side in order to enable the aggregation among different sources. Table 5.1 lists available aggregate functions. The user can also define an optional *groupingClass* field, which tells the platform how the aggregation should be partitioned. This grouping is made possible by the semantic model that is kept alive at runtime which identifies every resource and put it in relation with other resources. Therefore, a user can, for example, group by virtual machine and the resulting output will be one monitoring datum for each virtual machine. If no grouping class is specified, data coming from target resources will be aggregated all together. Finally, when the *aggregation* field is omitted, no aggregation will be performed and the rule will predicate on top of raw data, which means that the condition will be evaluated on every datum and actions will be executed for each datum.
- *condition*, an optional field, a boolean condition to be verified at every time step on either the result of the aggregation, in case it is specified, or else on raw data. It allows the user to tell the platform to check a condition and execute actions only if the condition is satisfied. If the condition is omitted, actions are executed at every iteration. The

condition is verified against the aggregated value, within each group (if the grouping class is specified) and at every evaluation of the rule, that is every *timeStep*. If no aggregation is specified the condition is verified for each datum. The condition should respect the following grammar:

```
<condition> ::= <term> | <term> '||' <condition>

<term> ::= <factor> | <factor> '&&' <term>

<factor> ::= <atom> | '!' <factor> | '(' <condition> ')'

<atom> ::= <var> <operator> <var>

<var> ::= 'METRIC' | <decimal>

<operator> ::= '>=' | '<=' | '=' | '<>' | '>' | '<'
```

- *actions*, a list of actions which should be executed at each evaluation step. If a condition is specified in the rule, the actions are executed only in case the condition is satisfied. We defined a general purpose action which is *OutputMetric*. This action simply allows to expose the resulting metric to metrics observers (see Section 6.5). The list of the actions that are currently implemented, together with a description of how new actions can be implemented is available in Section 6.4.2. In order to create parametrized actions, we defined two placeholder that can be used within actions parameters fields: *METRIC* and *RESOURCEID*. *METRIC* is the placeholder for the computed value. *RESOURCEID* is a placeholder for the id of the incoming monitoring datum, if no *aggregation* is specified, or the id of the resource for which the aggregation is grouped by, if *aggregation* and *groupingClass* are defined. If the aggregation is specified without any grouping class then *RESOURCEID* has no value and the user should put a hardcoded value.

5.3.1 Monitoring rules generation from QoS constraints

A set of monitoring rules is automatically generated starting from QoS constrained. Such rules will instruct the platform on how to generate events for such constraints in case they are violated. Such events can be observed by any interested metric observer. Monitoring rules are generated according to the following conventions:

1. the acceptable values or range for the constraints is negated and will constitute the condition under which the actions in the rule are executed;

2. an action that outputs a violation metric is defined by default, with the metric name being the name of the QoS constraint with the "_violation" suffix appended to it;
3. time windows is set to 60 second by default
4. time step is set to 10 seconds by default;
5. rules are enabled by default, which means that they will start being evaluated as soon as installed.

After the rules have been generated from QoS constraints, the user can modify the generated rules or create new ones.

5.4 Configuring data collectors

Once rules have been modeled we need to include in our system data collectors that can actually produce the metrics required by rules. There are very few requirements on how a data collector should be built. In general they can either be standalone components or a libraries.

For the purposes of the MODAClouds project we developed a library for monitoring Java applications that can provide application level metrics such as response time and throughput[1]. In order to monitor system level metrics a data collector was developed with the Imperial College London as our project partner[2]. This is a standalone Java application that requires to be running on every resource instance that needs to be monitored.

Data collectors are detailed in Section 6.4.1.

Data collectors will require at runtime information about the resource being monitored via environment variables. This information can be easily conveyed by configuration and management tools. Within the MODAClouds project, we used CloudML 4Clouds for the purpose and it was performed by attaching to the required components the following properties:

```
env:MODACLOUDS_TOWER4CLOUDS_VENDOR_ID = ${this.provider.id}
env:MODACLOUDS_TOWER4CLOUDS_EXTERNAL_VM_ID = ${this.host.name}
env:MODACLOUDS_TOWER4CLOUDS_EXTERNAL_VM_TYPE = ${this.host.type.name}
env:MODACLOUDS_TOWER4CLOUDS_INTERNAL_SERVICE_ID = ${this.id}
env:MODACLOUDS_TOWER4CLOUDS_INTERNAL_SERVICE_TYPE = ${this.type.name}
```

Data collectors require this information not only for sending the resource id within monitoring data but also to send the Tower 4Clouds server semantic information on how the resource being monitored is related to other components in the model.

[1] http://deib-polimi.github.io/tower4clouds/docs/data-collectors/java-app-dc.html

[2] https://github.com/imperial-modaclouds/modaclouds-data-collectors

CHAPTER 6

A Multi-Cloud Monitoring Platform

Tower 4Clouds is a multi-cloud monitoring platform specifically because it deals with both the heterogeneity and the ephemerality of multi-cloud applications. As we introduced in Chapter 4.1, a *runtime model* is maintained alive so that it represents the evolution of the system being monitored and allow the Data Analyzer to interpret monitoring data according to monitoring rules. Such runtime model is an instance of the application specific meta-model defined by the user starting from the base meta-model as we described in Section 5.1.

Tower 4Clouds is also highly extensible, making it easy to adapt to interoperate with multiple existing solutions.

Figure 6.1 depicts the runtime architecture of the platform and will be used as reference representation throughout the chapter. An overview of the components was provided in Section 4.1. In this section we are going to provide more details of the single components, how their functioning addresses the challenges brought by the multi-cloud environments and examples of implementations for both Data Collectors, Metrics Observers and Users. In Section 6.1 we provide details regarding the *Data Analyzer* and how preliminary results from the *Stream Reasoning* research area was exploited for the purposes of our approach. We then describe how ephemeral-

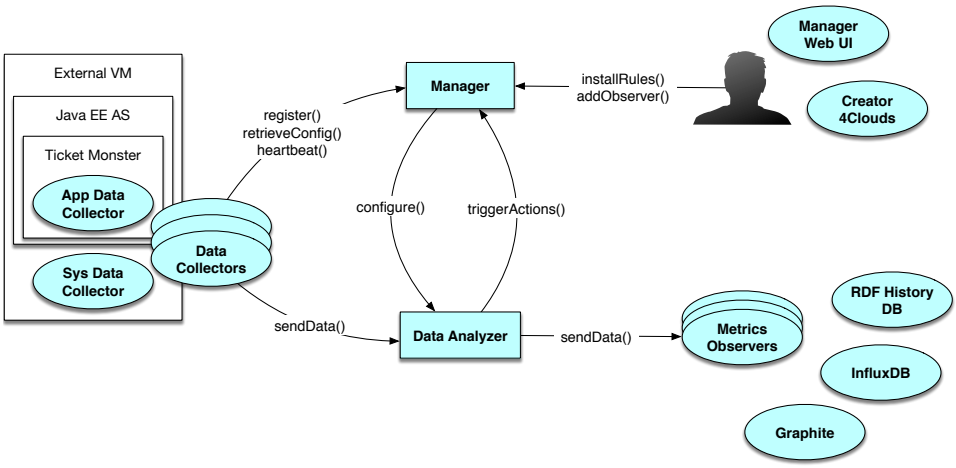

Figure 6.1: *Tower 4Clouds runtime architecture*

ity (Section 6.2), heterogeneity (Section 6.3) and extensibility (Section 6.4) are addressed. Finally, in Section 6.5 we show what Metrics Observers are and how existing tools can be attached to metrics.

6.1 A stream reasoner at the core

Given our modeling approach 5.1 and our monitoring rules language 5.3, the Data Analyzer had to address the following requirements:

- it had to be able to reason on top of both streaming (*i.e.*, monitoring data) and static knowledge (*i.e.*, the runtime model);
- it had to support inference rules in order to correctly interpret the *subclass* relation (*i.e.*, if a resource is an *InternalVM*, then it is also a *VM*) or transitive properties (*i.e.*, if a resource *X* is located in *Location A*, and *A* is located in *B*, then the resource is also located in *B*);
- it had to be able to perform statistic aggregations;
- it had to be able to verify conditions.

Stream reasoning is a novel research area which integrates results from both stream processing and semantic web areas [32]. Researcher in this area proposed a continuous query language (*i.e.*, *C-SPARQL*) and a tool (*i.e.*, *rsp-services-csparql*) for reasoning upon streaming information based on background knowledge [24]. Reasoning is performed upon information

structured according to the Resource Description Framework (RDF) language specification. Since both RDF and the C-SPARQL language were able to satisfy our requirements for data analysis, we adopted and extended the rsp-services-cpsarql so to integrate it as data analyzer of our Tower 4Clouds runtime monitoring platform.

Although C-SPARQL is a very powerful general purpose query language, it is too complex to be adopted by all developers and operators. The monitoring rules language we defined in Section 5.3 creates an abstraction layer specific for monitoring that is then translated into C-SPARQL queries by the Manager component. Data collectors specification is also creating an abstraction layer from RDF both for easing the creation of new Data Collectors and for reducing network traffic. In fact, RDF is a very expressive language, though verbose. Therefore, Data Collectors send monitoring data in a more compact JSON serialization format. The original rsp-services-csparql was then modified so to parse monitoring data in JSON format and convert into RDF format before processing happens. Moreover, information regarding the resources being monitored is also sent by Data Collectors in JSON format to the Manager when registered. The Manager is then responsible of updating the RDF knowledge base maintained locally by the Data Analyzer (*i.e.*, the rsp-services-csparql).

6.2 An elastic platform

In Section 2.5 we defined *elasticity* as the ability for a monitoring platform to continuously operate, without interruption and without any manual reconfiguration, in case of changes in the environments. Tower 4Clouds is able to address the intrinsic dynamicity of modern cloud applications by means of a passive discovery mechanism, where Data Collectors are responsible of updating the central runtime model with information regarding the resource they monitor. The configuration is provided by monitoring rules, which predicate on the design time meta-model of the system. This approach allows the monitoring platform to continuously operate, without reconfiguration.

The first implementation of the discovery mechanism was implemented by integrating the platform with CloudML 4Clouds [29]. A REST API was implemented and exposed by the Tower 4Clouds Manager to add and remove resources. CloudML 4Clouds, which already maintained its own version of runtime model of the deployed application, was extended so to actively notify Tower 4Clouds server when changes occurred. However, this implementation was preventing our approach to address the interoper-

ability requirement. In fact, it would be more challenging to integrate the platform with different discovery tools.

Therefore, in the final implementation, we moved the responsibility of updating Tower 4Clouds runtime model to Data Collectors. Data Collectors already implement a specification for sending metrics, and a Java library was developed, therefore it was more convenient to implement also this functionality on their side. Each Data Collector is responsible of updating the central runtime model with information regarding the resource it monitors by communicating it to the Manager. Data collectors will then notify their liveness via a heartbeat mechanism. If within the configured time to leave (TTL), no heartbeat is received by the Manager, respective resources will be deleted from the runtime model.

6.3 Designed for heterogeneous environments

While designing the communication protocol between the Data Collectors and the other components of the Tower 4Clouds platform we had to consider that not all cloud services offer the same features and flexibility. Multi-cloud applications are distributed across heterogeneous resources with different capabilities. For example, a PaaS solution may not allow a service to bind to a specific port, or else, a firewall in a IaaS cluster may block most of the incoming traffic. Therefore, in order for our monitoring platform to guarantee comprehensiveness (as defined in Section 2.5), the communication protocol had to minimize the requirements on the client-side. For this reason, our Data Collectors specification enforces a monodirectional communication protocol with both the Manager and the Data Analyzer. There are no requirements on Data Collectors host for listening to incoming connections.

Besides technological challenges, heterogeneity of cloud resources brings a complexity which may challenge the extensibility and maintainability of the system as far as monitoring is concerned. Describing heterogeneous resources with different hierarchies and relationships is easier with an underlying runtime model (Section 5.1), since it does not force users to hardcode the hierarchy into metrics names. The underlying meta-model can be extended whenever new technologies appear by adding new classes and new relationships. Old monitoring data will maintain their semantic meaning. Suppose, for example, that we were using the Graphite metric path to describe a monitoring datum regarding the response time of an application running on a virtual machine. The metric name would be something like: *vm1.app1.login.responsetime*. Then suppose that we want to change the

deployment of the application and run it into a Docker container instead of using virtual machines. We would need to update the metric schema, and the new metric would be: *vm1.container1.app1.login.responsetime*. Since the Graphite metric path hierarchy is merely based on ordering this would add complexity when it is needed to compare new metrics with old ones. Such problem is more recently solved by tagged based metrics, where meta-data schema does not rely on ordering. However, we believe the flexibility brought by RDF and our approach using a central runtime model is even more powerful:

- meta-data is not carried with the metric, therefore there is no need to agree on a common metric schema on the client side;
- new technologies can be integrated in existing meta-models and monitoring setup by using the inheritance provided by our framework: a *Container* service can be defined as a subclass of the *InternalVM* class, or a Amazon Lambda Function could be integrated as a subclass of both a *Method* and *ExternalComponent*, allowing existing rules targeting virtual machines and methods to seamless integrate these new technologies.

6.4 An extensible framework

Given the huge number of existing monitoring solutions today, it is fundamental for a new platform to be able to potentially interoperate with at least the most popular ones. In this section we overview the extensibility provided by our runtime platform by showing how new Data Collectors, new actions and new metrics observers can easily be implemented or integrated.

6.4.1 Implementing Data Collectors

New Data Collectors can be easily implemented by using the provided *data-collector-library*. The library offers a *DCAgent* class which is responsible of managing the communication with the server. Once started it will first take care of contacting the Manager, registering an object named *DCDescriptor* containing the information about the Data Collector, what metrics it is able to collect and containing the information of the resources it monitors. The *DCAgent* will take care of respecting the proposed specification, sending heartbeats to the Manager to notify about its health and check if the monitoring configuration was updated by the installation of new monitoring rules.

Here we list the main methods exposed by the *DCAgent* public interface:

```
public void setDCDescriptor(DCDescriptor dCDescriptor);

public boolean shouldMonitor(Resource resource,
  String metric);

public void addObserver(Observer observer);

public Map<String, String> getParameters(Resource resource,
  String metric);

public void send(Resource resource, String metric,
  Object value);
```

setDCDescriptor is used to configure the Data Collector. *shouldMonitor* is a method offered to ask if the Data Collector should send monitoring data regarding a specific resource and a specific metric. This method is used for example in one of the Data Collectors we implemented for monitoring the response time of a method. Whenever a method is called, if *shouldMonitor* is true, the data is sent otherwise nothing happens. In order for a Data Collector to be notified about any change in the monitoring configuration, it can register an observer through the *addObserver* method. When the *DCAgent*, after a synchronization with the server, identifies a change, all observers are notified. Note that here we are talking about the Java concept of observer, it does not related any how with the concept of monitoring observer we defined in Section 2.3. Through the method *getParameters* the Data Collector can extract the parameters that were specified by the user in the *metric* field of the installed monitoring rule. Such parameters could specify for example the sample time, or the sample probability. Finally, the *send* method is used to actually send a monitoring datum to the Data Analyzer.

Our *App Data Collector*[1] (depicted in Figure 6.1) exploits reflections and aspect oriented programming for collecting monitoring data from Java applications. The available implementation currently support the following metrics:

- *ResponseTime*: the interval between the invocation of a monitored method and the time when the method returns;

- *EffectiveResponseTime*: the response time minus the time spent for remote executions. Remote are explicitly decorated by the user via instrumentation.

[1] http://deib-polimi.github.io/tower4clouds/docs/data-collectors/java-app-dc.html

- *Throughput*: the rate of calls to a monitored method in number of requests per second.

The *Sys Data Collector*[2] (depicted in Figure 6.1) was developed together with the Imperial College London for the MODAClouds project. The following 12 different Data Collectors were included in a single agent able to offer all the metrics provided by these tools:

- JMX,
- Collectl,
- Sigar,
- MySQL,
- Log file parser,
- Flexiant Cloud monitor,
- Amazon EC2 CloudWatch,
- Cost monitor,
- Detailed cost monitor,
- Availability monitor,
- Start-up time monitor,
- EC2 spot price monitor,
- Haproxy log monitor.

Finally, a cluster level Data Collector was developed for demonstrating further the extensibility of our approach[3]. The integration experiment was conducted with Flexiant cloud provider which as a MODAClouds partner exposed APIs for retrieving monitoring data in form of CSV files. The following metrics are provided by the Data Collector (the first five referring to physical nodes:

- CPUUtilization;
- RamUsage;
- Load;

[2] https://github.com/imperial-modaclouds/modaclouds-data-collectors/wiki
[3] http://deib-polimi.github.io/tower4clouds/docs/data-collectors/flexiant-dc.html

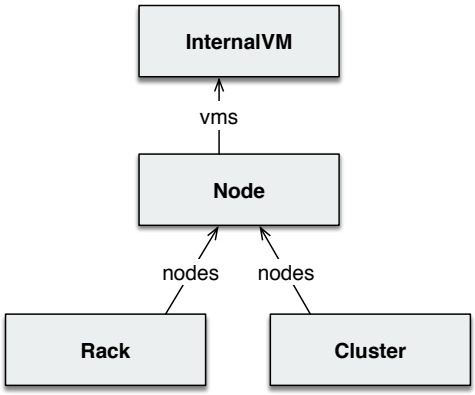

Figure 6.2: *Extension of the base meta-model required by the cluster level data collector*

- TXNetwork;
- RXNetwork;
- StorageCluster;
- RackLoad.

For the purpose the base meta-model (Section 5.1) was extended with the following classes (see Figure 6.2:

- *Node*, an internal component which contains multiple virtual machines;
- *Rack*, an internal component which contains multiple nodes;
- *Cluster*, an internal component which contains multiple nodes.

6.4.2 Implementing actions

New actions to be used in monitoring rules can be easily implemented in Tower 4Clouds by creating a new Java class with the desired name, extending the *AbstractAction* class. The abstract methods to be implemented by the new action are:

```
Set<String> getMyRequiredPars();

Map<String, String> getMyDefaultParameterValues();

Collection<? extends Problem> validate(MonitoringRule rule,
  List<MonitoringRule> otherRules);

void execute(String resourceId, String value,
  String timestamp);
```

The first method should return the set of parameters required by the action. The second should return the default values for each of the parameters. The *validate* method let the user implement any validation this new action should pass in order to be installed. The *rule* parameter in *validate* method is the current rule being installed, while *otherRules* are all rules that already installed. Given, for example, the *EnableRule* action implementation, *validate* is checking if the rule to be enabled is actually installed. Finally, the *execute* method is the code to be run whenever the condition is verified. This method is passed as parameter all the information regarding the aggregated monitoring datum computed by the rule in case it is required by the action.

Actions that were developed are listed in Table 6.1.

6.5 Metrics observer

Once a new metric is generated by means of an *OutputMetric* action in a monitoring rule, a metric observer can be attached. A metric observer is simply any service listening on a port to either HTTP, TCP or UDP connections, and able to parse any of the implemented serialization formats.

The implemented serialization formats that we implemented[4] in our Data Analyzer (*i.e.*, the *rsp-services-csparql* are the following:

- *RDF/JSON*, the default format provided by the *rsp-services-csparql*;
- *Graphite*, for sending metrics to the popular time series database Graphite (depicted in Figure 6.1);
- *TOWER/JSON*, a simple JSON serialization;
- *INFLUXDB*, for sending metrics to the more recent Influxdb time series database (depicted in Figure 6.1).

6.5.1 Saving historical data

Among the metrics observers, we implemented a service which records observed metrics as well as timed snapshots of the runtime model, the *RDF History DB*[5] (depicted in Figure 6.1).

Historical data is of high importance to be able to do *ex post* analysis. Deeper understanding of failures in the past can be obtained, identifying

[4] http://deib-polimi.github.io/tower4clouds/docs/data-analyzer/serialization-formats.html

[5] http://deib-polimi.github.io/tower4clouds/docs/observers/rdf-history-db/

Table 6.1: *Available actions (* = mandatory parameters)*

Action name	Description	Required Parameters
OutputMetric	Generate a new monitoring datum	*resourceId*: resource id name (default: RESOURCEID) *metric**: new metric name *value*: metric value (default: METRIC)
RestCall	Execute a rest call	*method**: {PUT,POST,GET,DELETE} *url**: the endpoint url *content*: the body of the request
EnableRule	Enable a rule	*id**: the id of the rule to enable
DisableRule	Disable a rule	*id**: the id of the rule to disable
CloudMLCall	Issues a command to CloudML 4Clouds, usually to perform a deployment change	*ip*: ip address of CloudML 4Clouds (default: 127.0.0.1) *port*: port of CloudML 4Clouds (default: 9030) *command**: command to be issued (possible values: SCALE, BURST) *type**: type of resource that will be considered *n*: number of instances to add (negative numbers allowed; default: 1) *cooldown*: seconds to wait before re-enabling the action (default: 600)

the root-cause and trying to avoid the problem in the future. Also, useful information on how to better exploit resource, or understand if a new release caused a performance degradation.

For this purpose, we created an observer able to store monitoring data in their native RDF format, together with their timestamp, in an RDF triple store (*e.g.*, Apache Fuseki). Monitoring data is periodically split into separate RDF graphs for performance reasons. Every graph has a timestamp with the time in which it was created and contains only monitoring data within the interval. Such interval can be configured based on the amount of data the platform is supposed to ingest.

However, monitoring data *per se* would not make any sense by itself, since we need the runtime model of the application to interpret it. We therefore saved the entire changelog of the runtime model as well. Deltas are stored by the Manager in a new graph every time a change is made, together with a timestamp. Periodically (1 hour by default), a complete snapshot is taken as well in order to construct the runtime model state at a given point in time faster.

We did not created a higher level language to query the historical data base, since historical analysis was not the main focus of this work, so SPARQL query language should be used.

CHAPTER 7

Evaluation

The objective of the evaluation is that of validating that our approach successfully addressed the challenges and relative requirements identified in Section 1.1, namely:

- *abstract from heterogeneity and prevent lock-in* (Requirement 1.1.1),
- *elastically adapt to ephemeral and dynamic systems* (Requirement 1.1.2),
- *limit the requirements on the data collector side to improve portability* (Requirement 1.1.3),
- *provide an extensible platform able to cope with future evolutions and interoperate with existing tools* (Requirement 1.1.4),
- *timely provide required information for reacting before end-user perception* (Requirement 1.1.5).

Evaluation for these requirements is mainly empirical, based on the experience of different users which tested and integrated our solution according to their business and technical requirements.

Then, we will run some experiments to evaluate the major current drawback of our solution which is strictly bounded to the current state of the art of RDF Stream Reasoners, which affects *scalability*.

Finally, we discuss about the remaining requirements listed in Section 2.5, how we addressed or could be addressed in a production ready solution.

7.1 Abstract from heterogeneity and prevent lock-in

The heterogeneity of today's hosting solutions and available services is addressed by the platform via a provider independent model based approach where complex systems can be modeled exploiting the runtime inference capabilities of our *Data Analyzer*. Inheritance allows to describe resources abstracting away from details and simplify user perception of the whole system. Monitoring rules address these classes, at any level, so that monitoring can be configured at the most appropriate level of abstraction. The underlying RDF model representation of the system enables multiple inheritance, so that a resource can be both an *Authentication Service*, an *Internal Component* and a *Java Application* and rules can target any of these classes. Moreover additional inference rules can be added to the knowledge base, such as transitive properties. We claim that provider independent modeling and the specified monitoring rule language able to target different level of abstraction guarantee the platform the flexibility to work in heterogeneous environments and prevents lock-in. To support our claim an example of heterogeneous deployment was shown in Section 4.3.2, where the Ticket Monster application was modeled, first, for a IaaS provider with qos constraints and rules. Then, the same application was modeled for a mixed IaaS and PaaS deployment. In the second case we were able to reuse most components and the same qos constraints and rules, with the exception of those targeting hidden resources (*i.e.*, PaaS infrastructure).

7.2 Elastically adapt to ephemeral and dynamic systems

In order to evaluate elasticity we had to verify that the platform was able to cope with both the cloud dynamism and the changes in monitoring requirements. We therefore first assessed that the runtime model was correctly updated while monitored resources changed. Once the model was updated, respective data collectors located on those resources were correctly configured and start monitoring. Next, we assessed that installing and removing rules at runtime worked properly and data collectors were correctly notified about the change of requirements. For our experiments we deployed both

Figure 7.1: *Model update time in seconds after starting one thousand virtual machines*

the *Manager* and the *Data Analyzer* on an AWS *t2.medium* virtual machine in Ireland.

In the first experiment we simulated the startup of a single virtual machine with a data collector deployed on it collecting the CPU utilization. Such virtual machine was running in Italy on a laptop, in order to simulate a multi cloud scenario where monitored resources can be located on different networks. The model was correctly updated with the new VM after 1180 ms after startup, 1030 ms being the network delay. In the second experiment we drastically increased the number of simulated virtual machines up to one thousand. Figure 7.1 shows the that we started all the machines in less than 15 seconds (a machine is started when the light gray bar begins) and it took around 50 seconds to have the model completely updated (which is when all the dark gray bars end). Currently, as soon as the Manager component receives a registration request from a Data Collector, the required update query is immediately created and performed to update the runtime

model. Model updates are serially executed. However, queries could be easily aggregated by the Manager, for example every second, to reduce the waiting time and increase performance. It is also noteworthy to consider that starting 1000 thousand machines at once is a very extreme case even for huge companies and a 50 seconds delay before starting to monitor can be ignored.

We ran another experiment to check if the model is updated correctly after resources expire and we shut down all one thousand machines at once. In such case, there is a configuration trade off between traffic cost and accuracy. An accurate model allows to have exactly the existing resources modeled and analysis is more performant compared to keeping all expired resources in the model. However, since the platform is able to work fine even if the model contains expired resources, we do not need to set the keep alive period to be very frequent. In our experiment we configured data collectors to synchronize with the server every 60 seconds. If no communication happens within 120 seconds, those resources are flagged as expired. Server side, a single thread checks every 60 seconds what are the expired resources and delete all of them in a single query. Therefore there is only one single call to the Data Analyzer which is able to remove all one thousand virtual machines from the model in less than a second when the query is issued.

Finally, we wanted to check if the platform is elastic when the monitoring configuration changes. We ran an experiment with fewer machines, just a hundred, in order not to incur in scalability problems and have this evaluation run in normal conditions. We installed a Monitoring Rule to ask all data collectors running on each virtual machine to send CPU utilization to the server and then uninstalled it after a hundred seconds. Figure 7.2 shows the results. At time 0 the rule is installed. Within 60 seconds all data collectors are configured and start sending metrics. After 100 seconds the rule is uninstalled and within 60 seconds all data collectors stop sending data.

7.3 Limit the requirements on the data collector side to improve portability

Data collector protocol requires a monodirectional communication towards the server. There is no need for a data collector to bind to a specific port, the responsibility is moved entirely to the client side. Data collectors autonomously connect to the server side for self-registering, communicating resources being monitored, retrieve the configuration, send monitoring data

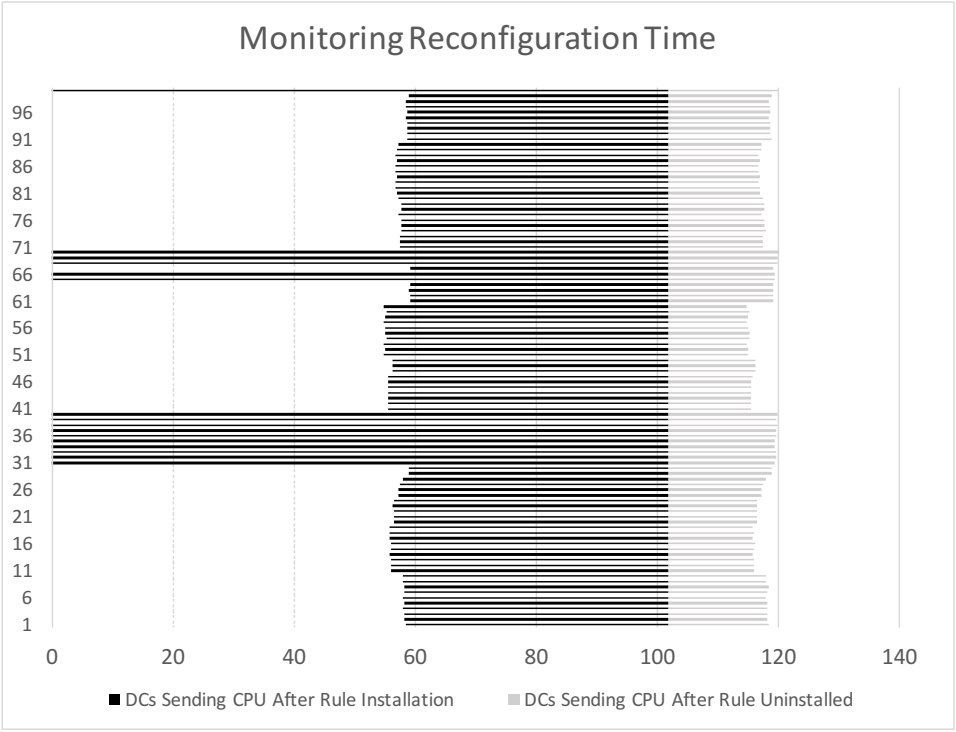

Figure 7.2: *Monitoring reconfiguration time for one hundred data collectors after rule installation and uninstallation*

and notify their liveness.

We claim this approach addresses the portability requirement since it is possible to push data even from PaaS solutions.

7.4 Provide an extensible platform able to cope with future evolutions and interoperate with existing tools

Extensibility and interoperability were validated empirically by asking several users, which did not know the platform in detail, to try to adapt and exploit Tower 4Clouds capabilities to address their case study requirements. We report here in this Section all users experiments and the feedback we received. Part of the evaluation is available in the MODAClouds public deliverable regarding the evaluation report [38].

7.4.1 Imperial College London

At Imperial College London[1], a PhD student managed to build several data collectors thanks to our data collector library. He first integrated Sigar and Collectl for host level data collection. Then created data collectors for collecting JVM metrics provided via JMX, MySQL metrics accessible via the SHOW GLOBAL STATUS query, application level metrics via an ad hoc log parser configured via a regular expression provided as parameter to the monitoring rule, Amazon Cloud Watch metrics and AWS costs via their public API, application and virtual machine availability via custom health checks.

After building these data collectors they wanted to exploit the platform to aggregate such monitoring metrics from all monitored resources and have them sent to a custom observer which was able to infer higher level knowledge via statistical analysis in order to obtain forecasts and correlations between metrics.

The interaction during the integration process did not require too many iterations. The provided Java library was easy to use to integrate all the existing agents with the platform and did not require more than one week of work.

The main benefit Tower 4Clouds offered to such integration was the aggregation, filtering and routing capability offered by Tower 4Clouds off the shelf. Thanks to the data collector library heterogeneous metrics coming from multiple data collectors were uniformed and made accessible via a single rule language.

[1] https://www.imperial.ac.uk

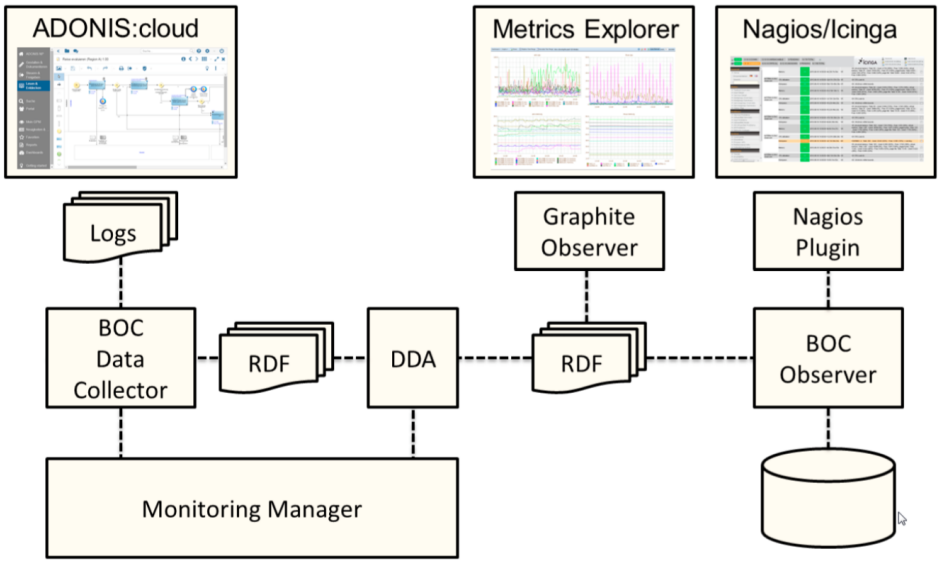

Figure 7.3: *BOC Group integrated monitoring solution*

7.4.2 BOC Group

BOC[2] has been using Nagios and Icinga for infrastructure monitoring for some years and its DevOps team has acquired the know-how required to use these tools. In the context of the MODAClouds European project they wanted to integrate the results of the Tower 4Clouds approach into their development and operational activities. They did not want to waste the expertise on the previous platform, therefore they decided to keep the Nagios and Icinga frontend interface, while integrating Tower 4Clouds data collection, transportation and processing mechanism. The key benefit they experienced from using Tower 4Clouds was the ease of creating new data collectors for extracting data from their application logs without any development effort, simply by configuring monitoring rules.

Another relevant benefit offered by Tower 4Clouds for BOC was the ability to use a single data collection technique that could then be processed and delivered to different tools (such as Nagios, Icinga or Graphite), without replicating the collectors.

Their integrated solution is depicted in Figure 7.3.

They were able to integrate the platform based on the provided documentation and it required only one hour of personal interaction with one of the employees and a couple of email exchanges.

[2] https://uk.boc-group.com

7.4.3 Softeam

Given the QoS constraint and the monitoring rule specifications and our monitoring rule library, Softeam[3] managed to integrate the modeling phase of our approach inside their modeling tool (Creator 4Clouds). Given our Manager API specification they also added a plugin in their IDE to allow the QoS engineer to publish the modeled monitoring rules to the running monitoring system by simply pushing a button.

They were also in the process of building a cloud version of their modeling IDE, namely Constellation, where they also implemented a custom dashboard which was easily attached as observer to Tower 4Clouds in order to show monitoring data about their tool when running.

Tower 4Clods was therefore integrated so that the monitored application was also providing a monitoring interface, proving the high interoperability of our platform.

7.4.4 Sintef

Given the specification of our data collectors, Sintef[4] was able to extend its deployment tool with the ability of defining environment variables that could be used to configure data collectors with information about the model when the application was deployed.

7.4.5 SeaClouds

SeaClouds[5] European project integrated Tower 4Clouds with Apache Brooklyn and used it as main monitoring platform in their project.

They did not used the entire feature set provided by our approach, however they automatized the deployment of data collectors based on the metrics defined on monitoring rules.

Moreover, they did not use UML as modeling specification but a simpler modeling language provided by Apache Brooklyn which is compliant with Tosca specification[6].

[3] http://www.softeam.com
[4] https://www.sintef.no
[5] http://www.seaclouds-project.eu
[6] https://www.oasis-open.org/committees/tosca/

7.5 Timely provision required information for reacting before end-user perception

Monitoring rules partially address the challenge by providing a configurable evaluation time step and time window. Our platform does not provide event based notification, however if it is important for a metric to be as fresh as possible, the time step can be setup even to 1 second and the rule would be evaluated every second. This would clearly impact scalability as detailed in Section 7.6 and tradeoffs must be made.

7.6 Scalability

There are still few works considering to use a stream reasoning approach as a possible solution to aid monitoring cloud applications and this is mainly due to the technological gap which prevents to use stream reasoners in the big scale. The expressiveness of the underlying query language and the ability to reason on both static and stream knowledge affects considerably the scalability. There are however ongoing studies on how to parallelize the computation to increase the scalability [41].

We wanted to assess how the current performance constraints affect the functionality of the current platform. Supposing we have R monitoring rules, each to be evaluated on average every S seconds, the average execution time T of each rule cannot be greater than $\frac{S}{R}$.

We first tested the execution time of a single monitoring rule aggregating the CPU utilization of an increasing number of virtual machines. The virtual machines were simulated, and their respective data collectors were sending a monitoring datum every 10 seconds. The rule had a time window of 60 seconds and was periodically executed every 60 seconds. Figure 7.4 shows the results. We can see that the execution time of a single rule increases exponentially with the number of monitored resources. When monitoring 1000 resources the average execution time is around 7.7 seconds, which means we have an upper limit of $\frac{60s}{7.7s} \approx 7.8$ installable monitoring rules of this kind. If we had 500 resources to be monitored the upper limit would raise up to $\frac{60s}{2.2s} \approx 27.3$ monitoring rules. These results clearly highlight that scalability is an issue for big systems and something to be addressed in the future work.

Finally, we tested the scalability of the execution time of monitoring rules, by keeping the number of monitored resources constant and increasing the number of rules installed simultaneously. We ran 3 experiments with 100, 250 and 300 rules uniformly distributed in time and with both

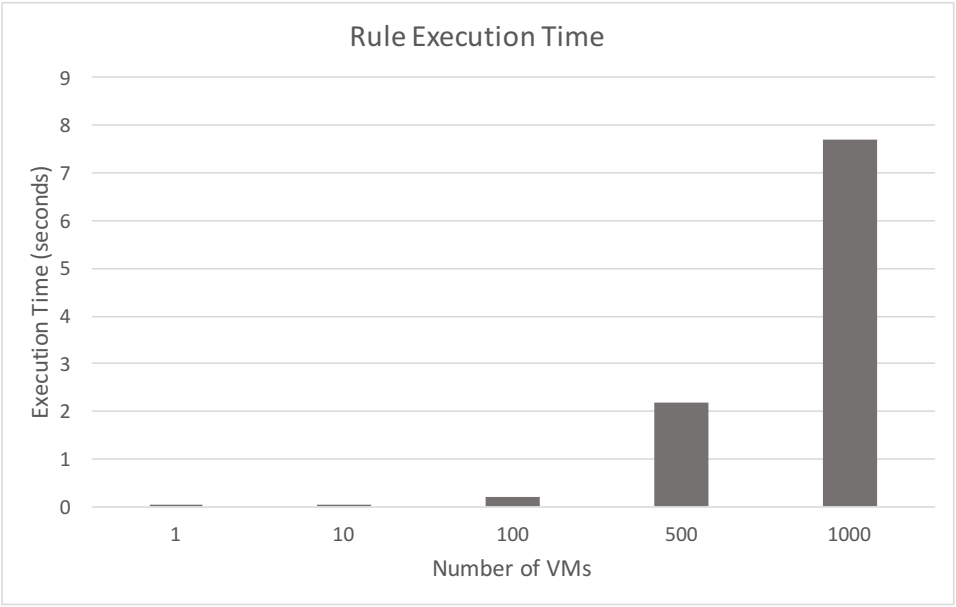

Figure 7.4: *Average rule execution time of a single monitoring rule with an increasing number of monitored virtual machines*

time step and time windows of 60 seconds, while monitoring the CPU utilization of 100 simulated virtual machines. The theoretical upper bound for such scenario, given the previous experiment where a rule predicating on the CPU utilization of 100 VMs had an average execution time around 0.2s, would be $\frac{60s}{0.2s} = 300$ rules. This result is reflected in the experiment shown in Figure 7.5, which highlight that when 300 rules are installed the platform saturates resources.

7.7 Other requirements

The implemented platform was built for the purposes of a research project so it is hard to evaluate *affordability* which is usually related to the pricing model of the monitoring solution. However, since our solution maintains the model alive at runtime, monitoring data does not have to carry the entire information about the resource but only its id, affordability was addressed by reducing the traffic load which is usually billed by cloud providers based on the size of transferred data.

Non-intrusiveness is clearly a requirement which depends on how each data collector is implemented. Famous and battle-tested tools we integrated such as Collectl and Sigar have very low overhead as they are written in C

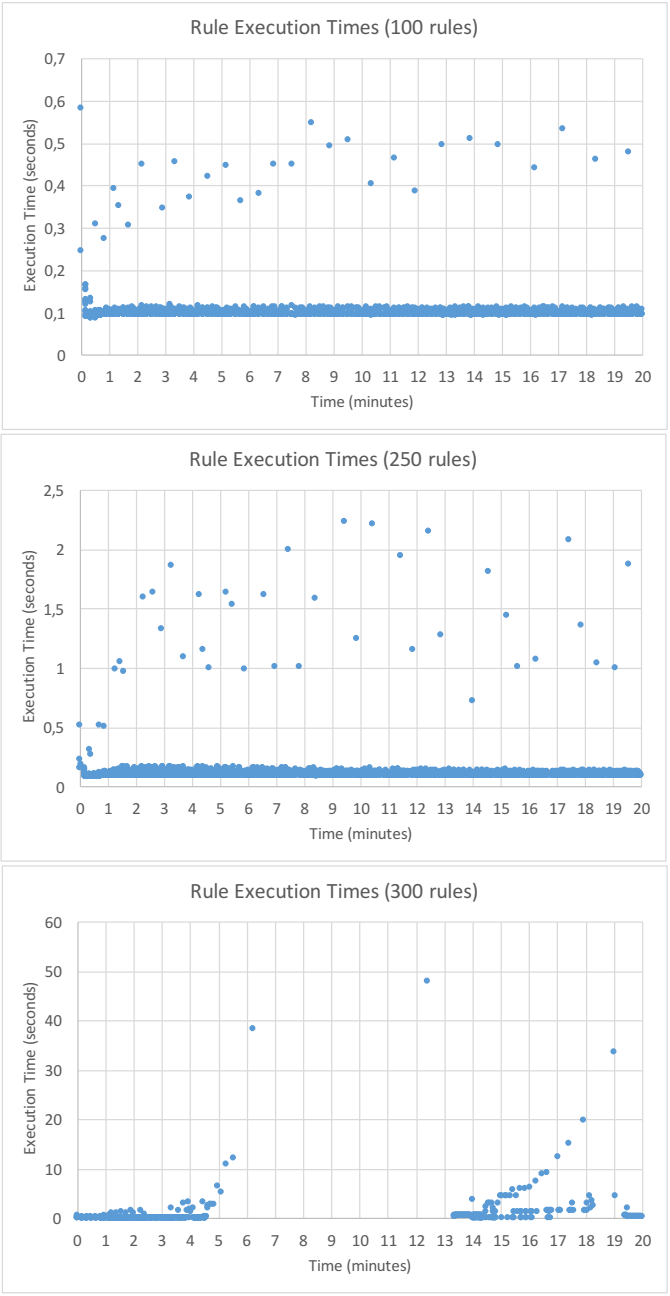

Figure 7.5: *Rules execution times over 20 minute experiment monitoring CPU utilization of 100 virtual machines installing 100, 250 and 300 rules with 60 seconds time window and 60 seconds time step*

and have been used and improved for many years[7]. The Java application level data collector we developed uses AspectJ which add negligible overheads[8] and its impact on normal functioning of the application depends on how the data collector developer use it. We avoided intrusiveness by limiting the number of synchronous operations and performing expensive operations, such as sending data to the data analyzer, asynchronously.

Multi-tenancy was partially addressed in our work. A data collector was developed to monitor also physical hosts on Flexiant Cloud through the API they exposed for us and by extending the meta-model with the new *Cluster*, *Rack* and *Node* classes. This extension allowed to aggregate monitoring data from multiple virtual machines by node. However the experiment was limited to only one cloud, and there was no information about tenants. In order to thoroughly address *multi-tenancy*, the base meta-model should be extended with a *Tenant* class and information regarding which resource belongs to which tenant should be provided to data collectors.

Adaptability is partially addressed. Monitoring rules actions can be used to change how data is collected in response to certain conditions. However this kind of adaptation is delegated to the QoS engineer and it is not automatically offered by the platform.

Autonomicity is addressed in the runtime platform thanks to the actions that the QoS engineer can set in monitoring rules. In the current implementation only one action for automatic remediation was implemented. The action allows to scale up or down the number of machines using the MODAClouds deployment tool CloudML 4Clouds in response to events. However, given the platform modularity, new actions can be added. It is worth noticing however, that the autonomicity that Tower 4Clouds can provide is only threshold based. In order to exploit more advanced techniques that involve machine learning algorithms, for example, ad hoc tools should be built and use the provided APIs to extract the required information from the platform, *i.e.*, attaching as observer to retrieve streaming monitoring data, query the model current status, query the history database via SPARQL queries.

At state of the art, there are several techniques used for production ready tools to address *resiliency*, *reliability* and *availability*. However, they were not implemented since it was out of the scope of this work. For example, *availability* could be addressed by creating a master-slave architecture on the server-side, replicating both the manager and the data analyzer. The model is slowly changing so it is easy to synchronize multiple instances

[7] `http://collectl.sourceforge.net/Performance.html`

[8] `http://www.eclipse.org/aspectj/doc/released/faq.php#q:effectonperformance`

of replicated stores. State of the art tools for service discovery (such as Consul [4] or ZooKeeper [17]) can be used to make data collectors communicate with the master server.

Finally, *accuracy* is not directly addressed since the focus was on monitoring definition and the platform and not on data collection methods and their accuracy.

7.8 Threats to Validity

The Tower 4Clouds approach aims at raising the attention to QoS monitoring since the first phases of the development in order to improve the feedback loop since the first runtime deployments. Besides the performance and scalability issues we investigated in this chapter we believe the major threat to validity for the approach is the applicability of such an approach to a real industrial scenario. Current release cycles are very demanding in terms of time to market and most of the software companies do not spend time modeling their system or are not aware of the QoS requirements since the first phases of development.

As we presented in this evaluation chapter, feedback was received from use case providers which participated to the MODAClouds research project. However, the feedback is limited to the applicability of the approach to prototypes and not to real world scenarios.

CHAPTER 8

The State of Practice: an Industrial Survey

In this chapter we describe the empirical study we conducted to discover the challenges practitioners are addressing by using existing monitoring solutions.

The main objective of the study is to identify what are the main challenges that industries are facing today while monitoring their software systems and how they handle incidents. For the purpose, a survey was conducted in order to assess tooling and methodologies currently used by industries to monitor their IT systems, understanding what people, tools and processes are involved in observing and managing both the quality of service and the experience perceived by users and to collect the open challenges.

Finally we aim at verifying whether the Tower 4Clouds approach can help satisfying some of the identified challenges.

8.1 Research Questions

Given the challenges and the importance of monitoring, this empirical study aims at understanding the following questions in today's companies:

- **Q1.** Is monitoring actually perceived as a fundamental asset?

- **Q2.** To what extent do companies monitor their software systems and how?
- **Q3.** How are incidents discovered and handled?
- **Q4.** What are the roles of people involved with monitoring?
- **Q5.** What are the most critical challenges perceived when trying to make a system observable?
- **Q6.** Does Tower 4Clouds provide any contribution that may help companies?

Asking to a broad sample of people to understand, try and test Tower 4Clouds would be unfeasible and we would have reduced the participation. Therefore, question Q6 was not explicitly asked in this research study. We tried to answer ourself after analyzing the results.

8.2 Research methods and approach

In this study we aimed at having a complete and comprehensive overview of the the practical use of monitoring in the industries. Therefore, there was the need for reaching the biggest number of companies across several areas and sizes. We decided to build a standardized form with closed questions to be shared across several broadcasting channels. The target goal was to gather feedback from at least 100 companies.

Research questions in Section 8.1 are very broad questions and required to be detailed in more specific and closed questions. However, there was little knowledge about the actual perception of today's industries about the matter. Therefore, with the intent of being as unbiased as possible, the study was conducted iteratively, starting from the main research questions, running few in-person interviews and gradually extracting the most valuable questions for the final survey.

The complete study design is here described in detail:

1. Starting from the above research questions (Q1-Q6), a list of open questions was prepared to be used as starting point for discussion. The purpose of these questions was to guide the discussion to understand the company main business, the role of the interviewee within the company and their software product. Then, they would help the interview to go through the practices, tools and people involved in the operations, maintenance and incident handling. Finally, they had to guide the discussion to have the interviewees expose problems and

open points regarding the available monitoring tools and practices. The fifteen questions that were prepared are reported in Appendix B.1.

2. For the first phase we aimed at selecting companies developing and maintaining software systems covering the following architectural styles:

   - *microservices architectures*, since they are an increasingly popular development approach and monitoring is challenging because of their intrinsic distributed nature;
   - *multi-tenant hosting*, since companies offering multi-tenant solutions need to take special care of monitoring in order to guarantee the SLA agreed with the different users when resources are shared among them;
   - *sensor networks*, since they have particular requirements such as energy efficiency constraints and unreliable infrastructures, requiring special monitoring attention.

   Moreover, companies were chosen also to cover the following areas that are orthogonal to the previous classification:

   - *web applications*, where monitoring is fundamental to provide required quality of service to users, which can cope with often unpredictable traffic spikes;
   - *real-time services* (*e.g.*, gaming or trading), where timing constraints need to be monitored;
   - *technology providers*, which are offering software solutions, installed on customers machines, and may provide support for monitoring on heterogeneous hosting solutions.

   Among the fourteen contacts we initially identified, seven accepted to participate. Among the selected architectures, sensor networks was not covered as well as real time services from the selected areas.

3. The fifteen questions were shared with the contacts together with the invitation to participate in the study. This allowed them to better identify the most qualified person for the interview and to gather information in advance if needed.

4. In-person interviews were scheduled and took about one hour each. When face to face meetings were not possible for time or spatial constraints, conference calls were organized. Anonymized summaries are reported in Appendix B.2.

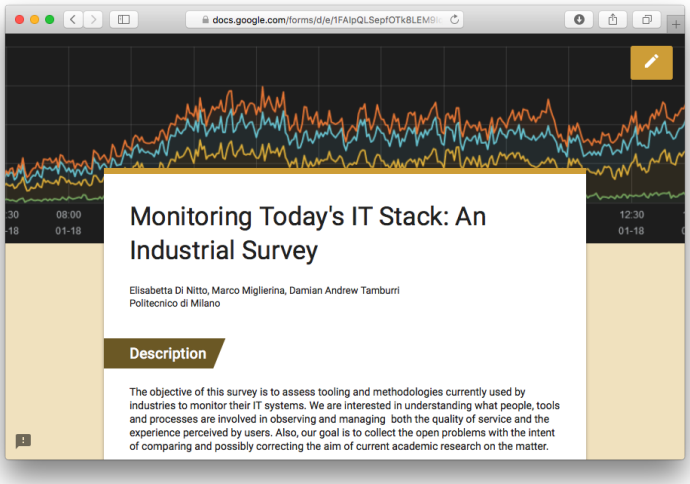

Figure 8.1: *Screenshot of the Survey online form.*

5. Interviews were analyzed in order to obtain first exploratory feedback and hypothetical answers to the main research questions (Q1-Q6). This analysis allowed to better identify closed questions and hypothesis to be verified across a bigger set of candidates during the second phase.

6. An online form with closed answers was prepared (Figure 8.1). The survey consisted of three sections: the first asking for information about the respondent role and the company; the second focused on understanding the software system architecture, its development and its releasing process; the last section about how monitoring is done and how incidents are solved.

7. The form was perfected iteratively in order to improve clarity and avoid ambiguities. First, it was shared among some colleagues of my research group, which were asked to review it by trying to fill it and provide feedback. Then, it was shared with some colleagues outside my research group for a further review process. Finally, the survey was sent to the people that were involved in the first phase of in-person interviews to begin collecting answers and check the first answer set and any provided comment or question to further verify possible problems.

8. The final form consisted of 38 questions and was prepared using a

Google Form[1]. The estimate time for answering was estimated to be 15 minutes. 141 Responses were collected by sharing the submission link via multiple broadcasting channels:

- personal contacts or their acquaintances;
- posts on social medias such as Facebook and Twitter;
- requests to advertise the form via mailing lists and communities (*e.g.*, we asked several meetup organizers around the world, which are focused on monitoring, to share the survey among their participants);
- posts on Reddit, under `/r/sysadmin` and `/r/devops` subreddits.

The form was left open for answers for around two months.

8.3 In-person interviews results

Table 8.1 summarize the sample that was considered for in-person interviews.

These interviews were useful to gather an initial feedback on monitoring in the industries. We started from these responses to collect a first set of considerations to be verified on a larger sample during the second phase. Insights will reference research questions listed in Section 8.1 (Q1-Q6).

First, the interest in monitoring exceeded our expectation (Q1). All interviewees considered monitoring a fundamental practice. It is performed in different ways, but all of them are doing it (Q2) and, most of all, no one asserted that monitoring is useless or that is not worth investing on it.

Small companies usually prefer cloud hosting solutions and tend to experiment with more recent technologies (*e.g.*, Docker containers) and patterns (*e.g.*, microservices). There is usually one team taking care of both development and operations (Q4). They do not invest much in monitoring, rather they prefer to use solutions that are available at small cost and without implying excessive effort (Q2). Incident handling is mainly diagnosed by means of manual log analysis (Q3). The most important metrics are availability and their product-specific business metrics.

In *medium companies* developers and operators are usually separated teams which have different responsibilities. Usually only operators receive alerts from the monitoring system (Q4). Operators are skilled enough to setup a monitoring platform using one or more open source monitoring

[1] `https://goo.gl/forms/UHY1PUEXVm64yhKF3`

Table 8.1: *Interviewees sample. Clarification: when a company has multiple ongoing projects the study refer to a single project. IT headcount is the number of IT people working on the selected product.*

ID	Domain	Role	Total/IT headcount	Product	Architecture	Software stack	Monitoring stack
I1	Travel	CTO	100/10	B2B and B2C web services	Microservices	Scala/Java, Akka, Docker containers, Amazon EC2	Amazon Cloud Watch, custom availability checks
I2	Software	Research Software Engineer in Innovation Lab	100/5	Stream processing tool for news validation (prototype)	Event driven, microservices	Java, MongoDB	Custom code instrumentation
I3	Software	System Architect	200/150	SaaS services for enterprises business management	3-tier, multi-tenant DB	Java, Tomcat, C++, SQL Server, HAProxy, Linux/Windows, third party IaaS	Focused on DB metrics, Graphite, Grafana, Nagios, Icinga, ELK stack
I4	Software	Chairman, CTO	5/3	SaaS recommending system for business management	monolith with Multi-tenant DB	Java, Google App Engine, Google SQL	Focused on DB and business metrics, Google Analytics, Google App Engine Dashboard
I5	Software / Hardware	Distinguished Engineer	~400.000	Enterprise infrastructure system	Mainframe solution	N/A	Proprietary monitoring tools, analysis and alerting tools for all IaaS and PaaS services they provide
I6	Software / Hardware	Solution architect	~100.000	N/A (general interview regarding multiple customers)	N/A	N/A	Financial corporations buy expensive and sophisticate on premise monitoring tools such as Dynatrace. Other companies prefer monitoring as a service with small setup efforts. Most companies use custom solutions with general purpose and already known tools such as relational databases and spreadsheets for data analysis
I7	Finance	CTO	40/5	e-payment platform	Microservices	Java, third party services (*e.g.*, Google API), relational database, Amazon S3, Amazon EC2, Amazon SNS	Amazon Cloud Watch, manual log checking (soon moving to ELK stack), custom library pushing metrics to Cloud Watch and/or to report table

tools for both system and application metrics and metrics can be shown on dashboards, usually accessed by operators only. Logs are collected and accessed using the ELK stack (Q2).

Large companies usually manage more complex systems and therefore tend to invest in more sophisticated monitoring solutions. Also, companies where incidents can cause huge losses prefer to pay for monitoring solutions and related support. These expensive solutions allow them to use advanced machine learning and big data techniques for analyzing monitoring data (Q2).

In all companies, alerts are mainly sent via email or via SMS in case of critical issues (Q3). If there is any automatic remediation based on some monitoring check, it involved only restarting a service or scaling up and down hosts (Q3). Issue detection is mainly based on threshold, only companies with strong IT departments or those investing a lot for third party monitoring tools have more sophisticated machine learning tools for detecting problems (Q2, Q3). In most of the cases custom code is used and monitoring data is analyzed using tools that are not monitoring-specific like relational databases or spreadsheets (Q2).

During the interview I3 a limitation of today's monitoring tools is identified, which is the lack of standards and integration. The company use different data collecting tools on the same machines with different scopes, with data flowing through different paths according to different protocols. Some metrics even overlap between each other.

This is for example a challenge addressed by Tower 4Clouds (Q6) since it enforces the usage of the same protocol for sending data to one single data analyzer which is then able to route processed data to different destinations according to monitoring rules.

Another challenge that was identified during this first phase is the high market demand and competitiveness. Startups (interview I7) or emerging companies (interview I1) that are trying to address new market opportunities, require a speed to implement new features that delays the application of quality assurance techniques unless they require very small setup efforts (interview I6).

8.4 Survey

8.4.1 Data acquisition

During the two months of data acquisition the responses trend was monitored in order to understand what were the most impactful communication

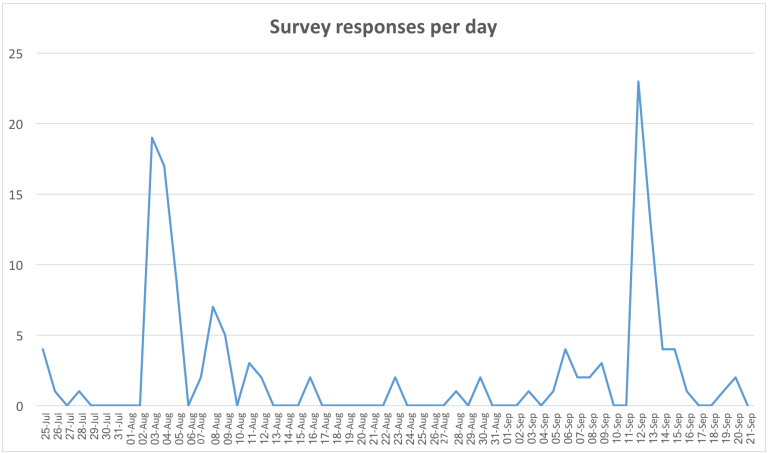

Figure 8.2: *Survey responses rate over time.*

techniques as a parallel study and the resulting trend is depicted in Figure 8.2. The trend clearly highlights three spikes that are related to two different advertising actions. The first, in the beginning of August, when the message was broadcast to around 250 personal contacts, requesting them to share the research survey to any of their acquaintance working in IT. The second a couple of days later when the same message was broadcast to around 100 other personal contacts. The third in the middle of September when the research was advertised on Reddit under the `/r/sysadmin` subreddit, which counted around 150.000 readers and revealed to be the most impactful channel.

8.4.2 Data sampling

After two months around 141 responses were collected. The sample revealed to be quite distributed in terms of industries types as can be seen in Figure 8.3. 43.7% belongs to technological industries. The rest is distributed among several areas: financial services, telecommunications, health care, consumer care and others.

Figure shows that the sample is well distributed with reference to the size of the companies. 25% are under 20 employees, 21% are between 21 and 100, 22% between 101 and 500, 22% between 501 and 10.000 and the remaining 10% over 10.000.

In terms of IT department size (Figure 8.5) 30% of the companies has less than 5 IT employees, 26% between 6 and 20, 18% between 21 and 100, the remaining 26% has an IT bigger than 100 employees.

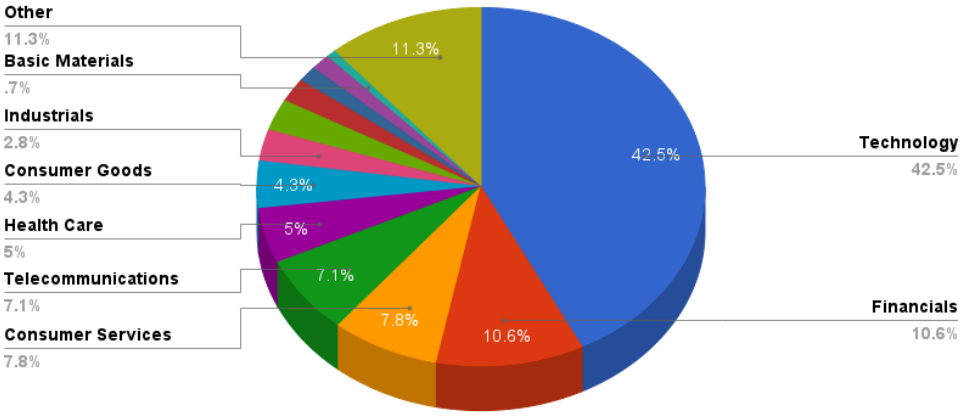

Figure 8.3: *Industry types distribution.*

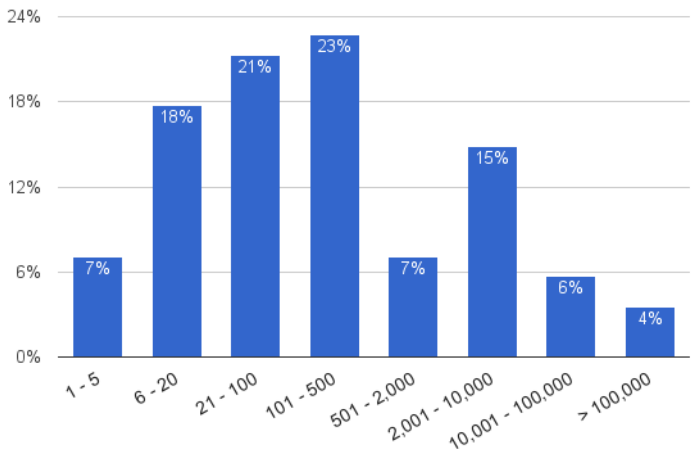

Figure 8.4: *Industry sizes distribution.*

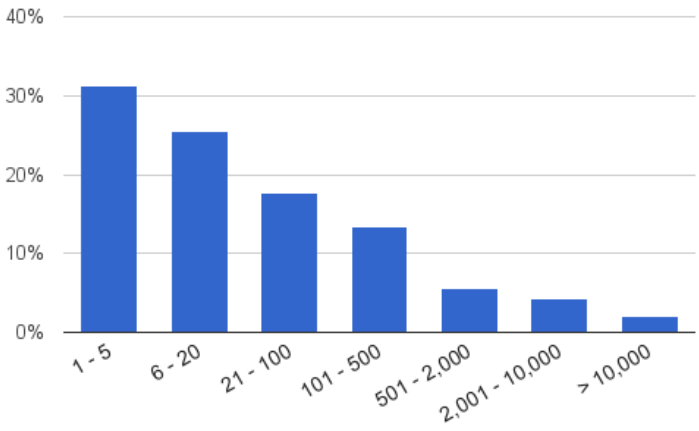

Figure 8.5: *IT sizes distribution.*

Respondents were usually covering multiple roles, however 96% of them were either developers, system administrator or devops engineer. Among these, 39% covered exclusively a development role, 14% exclusively a system administration role, 12% exclusively a devops role.

Software systems are distributed as depicted in Figure 8.6, more than a quarter of the companies develop enterprise software.

Architectural choices distribution is depicted in Figure 8.7.

Most software systems are composed of a small number of deployable components: 72% have less then 10 components, with 38% having less than 3 components. 20% have between 11 and 50 components, the remaining 8% over 50 components.

Figure 8.8 depicts the hosting solutions adopted by the interviewees. 36.17% of them use the public cloud, 11.4% of which using it as the only hosting solution for their entire system.

Regarding the deployment solution there is a considerable percentage of companies that deploy on bare metal and on IaaS (Figure 8.9). However a good sample of companies using also more ephemeral solutions or higher abstraction layers took part to the survey.

We also collected a subjective score of the automation level of the release cycle (Figure 8.10), the release rate 8.11 and, finally, the average workload their systems undergo (Figure 8.12). Only 4 companies release more than 10 times per day. These are large companies, two of them having more than 100.000 employees. They all use SOA architectures with hundreds of services, serving thousands of requests per seconds. Only one company releases less than once a year. It is a medium technological com-

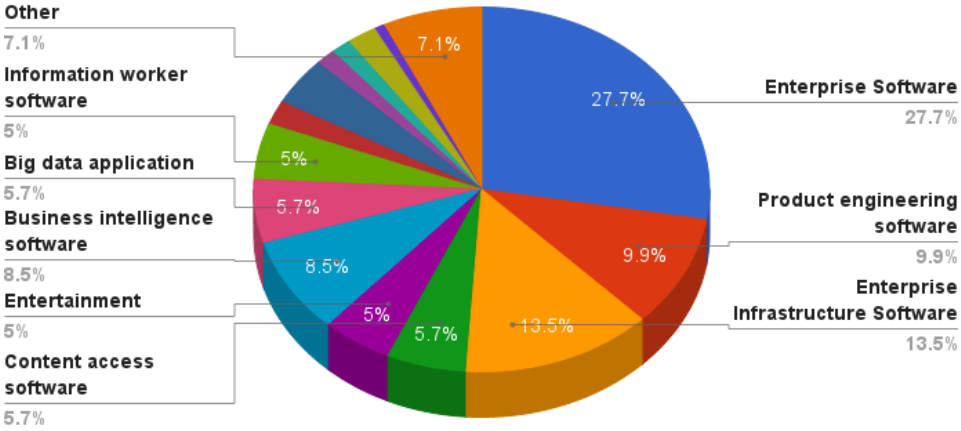

Figure 8.6: *Software systems distribution.*

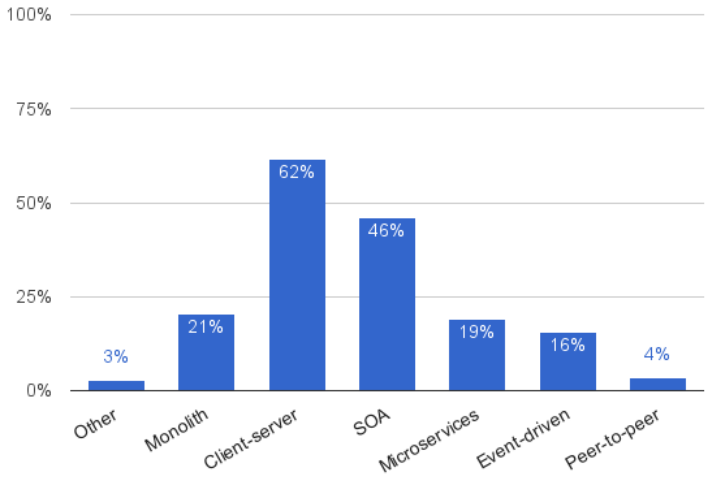

Figure 8.7: *Software architectures distribution. Clarification: a system can use multiple architectures*

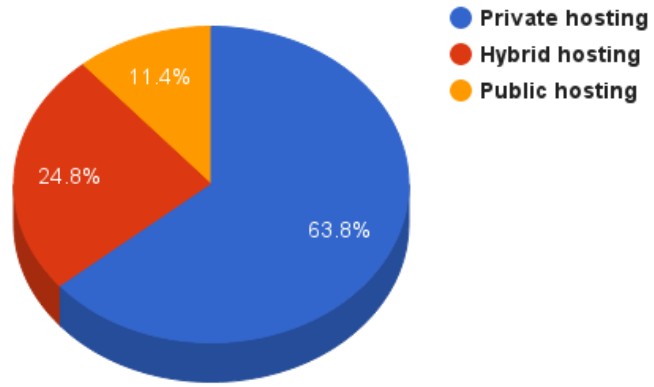

Figure 8.8: *Hosting solutions distribution.*

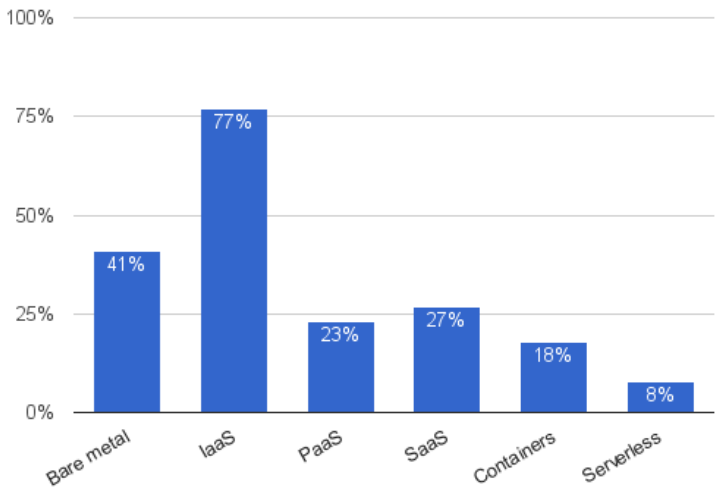

Figure 8.9: *Deploying model distribution.*

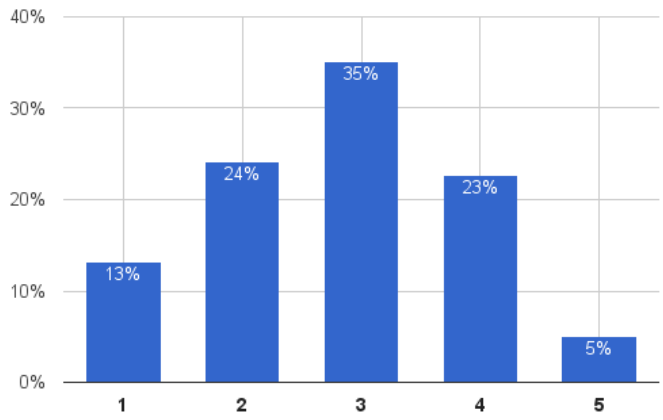

Figure 8.10: *Rates distribution for the level of automation in the release cycle. Average score is 2.83.*

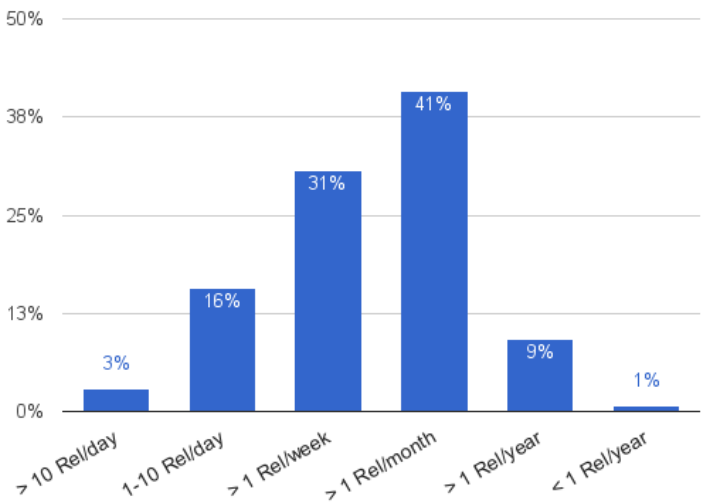

Figure 8.11: *Release rates distribution.*

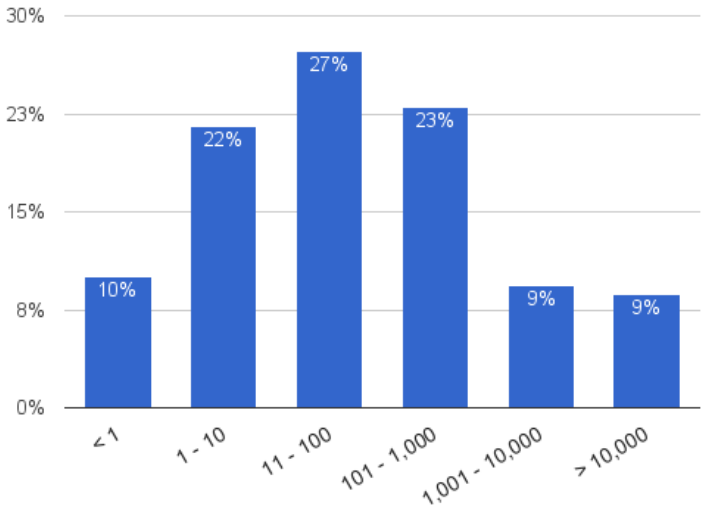

Figure 8.12: *Workload distribution in terms of users requests per second.*

pany with a small IT following a relatively big system composed of 20 to 50 components, privately hosted, serving hundreds of requests per second.

8.4.3 Results

Results given from the third section are going to be analyzed by considering each of the research questions listed in Section 8.1.

Q1. Is monitoring actually perceived as a fundamental asset? The large participation to the survey, even though the number of questions was high, is a first indicator of the interest that monitoring has across the industries. Nevertheless, when the respondents were asked whether they had planned to improve their monitoring asset (question C21), 12.8% of them answered *"No, because investing time and money on monitoring is not perceived as a profitable investment"*. According to our data, this perception can be found more concentrated in companies

- managing a smaller number of components,
- hosted on private clouds,
- with smaller automation processes in action and
- releasing less often.

These companies also resulted to give the observability of their system a lower score (2.59) than the average of the entire sample (3.1). From the answers they gave, these companies experience longer unavailability time on average, when incidents occur:

- more than 20 minutes in 84% of such companies, against 63% of the entire sample;
- more than 1 hour in 39% of such companies, against 28% of the entire sample.

Also, the average time to diagnose the cause of an incident is negatively impacted in these set of companies which do not consider monitoring worth the investment: only 28% of them are able to diagnose a problem in less than 20 minutes, against the 44% of the entire sample.

Q2. Do all companies monitor their software systems and how? 18% of the surveyed companies do not have any deployed system for monitoring, they are used to manually check via terminal commands such as *ping*, *ssh* or *grep* if there is any problem and why. These companies have usually small systems: 60% of them manage between 1 and 3 components, 32% between 4 and 10. They also have slow release rates compared to companies that practice continuous delivery. In fact, no one releases more than once a day. However, such companies are subject to small workloads and together with the limited complexity of the system they manage to have fewer incidents on average. Nonetheless, when incidents occur they usually experience longer unavailability time.

Moreover, 38% of the entire sample use no third-party monitoring tools. They only use internally developed solutions: besides manual inspection, some a third of these companies develops a custom dashboard.

Among the companies using third party tools for monitoring purposes, the most used ones are: Google Analytics (18%), Nagios (16%), MySQL (14%), Grafana (12%), Amazon CloudWatch (11%). However the monitoring tools offer is really fragmented among both commercial and open source solutions, with no clear winner.

Finally, it is worth considering, that almost 50% of all the surveyed companies release new features or new components with monitoring in mind, releasing new monitoring configurations or features specific for the new piece of the system.

Q3. How are incidents discovered and handled? 70% of surveyed companies use emails among the alerting techniques. 57% discover problems

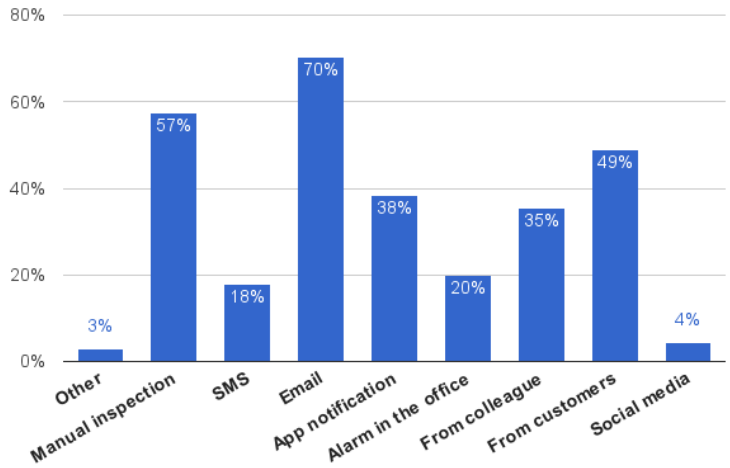

Figure 8.13: *Distribution of the techniques in place to discover incidents in companies. Interviewee could select multiple answers if multiple techniques were used.*

by actively inspecting logs or graphs. The third most frequent way of how employee are alerted is directly from customers. The complete distribution is depicted in Figure 8.13 The release of buggy core components is considered the most common cause for an incident by 38% of the respondents. The distribution of the other causes is depicted in Figure 8.14. The release of buggy core components requires manual intervention for 60.1% of the companies. For about half of the sample, manual intervention is also required for wrong configuration updates, filling of the disks and hardware damage. Automatic remediation (or partial mitigation) is instead common for about 40% of the cases in the case of CPU or memory resource saturation.

Q4. What are the people involved with monitoring? Monitoring was once considered an operator business. However data shows that this trend is changed since in about 63% of the companies developers access monitoring information and can receive alerts. In 16% of the cases also the business access monitoring information and, differently from our expectation, in 31.06% of the companies business can receive alerts.

Q5. What are the most critical challenges perceived when trying to make a system observable? The last question in the survey asked the interviewees what were the main obstacles they perceived to be preventing the adoption of monitoring. Figure 8.15 shows the results. 50% of the in-

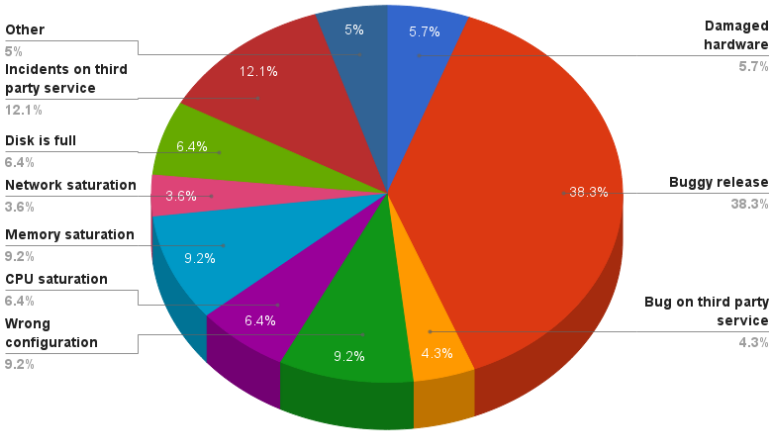

Figure 8.14: *Distribution of the most common cause for incidents according to interviewees.*

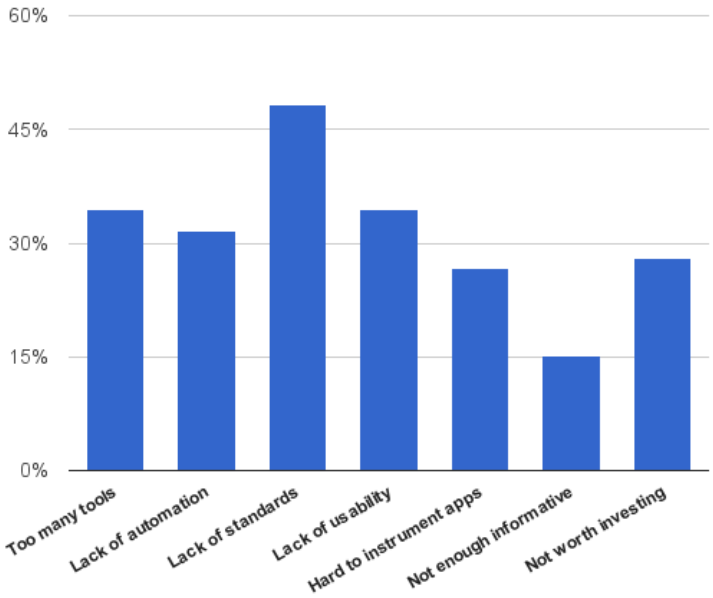

Figure 8.15: *Distribution of challenges perceived by interviewees in the adoption of monitoring.*

terviewees perceive the lack of standards as the main obstacle. This aspect, together with the proliferation of tools, each one bringing new protocols and schemas, is undoubtedly one of the most critical challenges. Last, they are hard to use, which actually supports what the in-person interviews highlighted, that is, self-service and easy to setup solutions are preferred.

Q6. Does Tower 4Clouds provide any contribution that may help companies? In order to address the proliferation of tools it is fundamental at least to provide extensible platforms, so that new adopters can integrate some of their tooling and reuse part of the acquired skills. Even though this does not solve the problem, it is a requirement almost all most recent tools (*e.g.*, Sensu or Grafana) dealt with, and Tower 4Clouds is built to be extensible as well.

Probably, the main contribution that Tower 4Clouds brings to the challenges highlighted during this study is towards standardization. In fact, even though we provide yet again a new communication protocol and a new way of describing data, our approach could provide hints to the definition of a new standard, at least within the same organization:

- The definition of a base meta-model which is proposed with the approach is creating a shared ontology which can be cooperatively extended;
- within an organization the explicit definition of a domain specific model which is then used by the monitoring platform is another step towards standardization of names and relations within the same company.

In modern companies, developing using microservices architectures, with one team responsible for each microservice, monitoring is an orthogonal tool, used and required by all teams. This is why it is so important to have shared vocabulary.

8.5 Discussion

This survey helped identifying the state of practice and the main challenges industries are facing when approaching monitoring. The main challenges that monitoring tools users encounter can be summarized as:

- *C1. Lack of standards.* There is no standard protocol for sending monitoring data and no standard schema for declaring them.

- *C2. Strong dependencies among teams.* Monitoring is a support service, required by all other services composing the system. Changes to the monitoring platform affect all other teams focusing on those services.
- *C3. Fast evolution and tools proliferation*. There are new technologies and new environments coming out every other day, together with a new monitoring tool.
- *C4. Lack of reusability*. Each company creating its preferred monitoring infrastructure is implementing its own automation scripts for deploying and configuring it.

To our knowledge, these challenges are not yet addressed by any research approach and we tried to address them in a recent work we described in Chapter 9.

CHAPTER 9

Towards Omnia: a Monitoring Factory for Quality-Aware DevOps

In Chapter 8 we observed that effective monitoring is still a difficult task, approached in several heterogeneous ways. Although a huge number of monitoring tools, both commercial and open-source ones, proliferated in the last few years, there is no holistic framework that drives the embracing of standardized solution for monitoring.

The monitoring platform we developed for this thesis, Tower 4Clouds, introduces a novel approach where quality of service is given high relevance since the first design phases. However, the approach is subject to the limitations discussed in Chapter 7. The platform requires a high learning curve and may not be applicable to industrial scenarios as is.

The main objective of this chapter is to present an initial investigation of a monitoring solution, by offering an approach called *Omnia*, whose key objective is reducing the learning curve and entry-cost to monitoring technologies.

Omnia is an approach that assists system administrators in deploying a monitoring system and developers in configuring and accessing monitoring information, exploiting DevOps practices such as *infrastructure-as-*

code (IasC) and *automation* [55]. Omnia consists of two major parts: (1) a *monitoring interface* for developers that helps using monitoring systems, independently of the specific implementation, and (2) a *monitoring factory* for system administrators that helps building a monitoring system that is compatible with such interface, leveraging existing monitoring tools. Our approach is a reinterpretation of the factory pattern [39]. Similarly to the famous design pattern, our *monitoring factory* creates a concrete implementation of a monitoring system (by automatically composing and configuring existing monitoring tools) and users refer to it using a common *monitoring interface* that is independent of the actual implementation.

Together with the implementation of the tool, we also propose the definition of a common reference vocabulary for resources being monitored and general purpose metrics, versioned with the Omnia source code and to which every component should adhere when integrated with our tool. Although many research works exist in the scope of monitoring research (*e.g.*, as highlighted by several surveys in the field [22,35]), a fundamental lack in the monitoring research scenario is the definition of a reference dictionary.

On the one hand, the approach proposed in this chapter helps system administrators address the multitude of available tools and easily setup a monitoring system. On the other hand, *Omnia* assists all software practitioners throughout all phases of their monitoring infrastructure life cycle (*e.g.*, dashboard configuration, data exchange, analytics representation and more) providing to the entire organizational structure a single protocol, a common vocabulary and a versionable monitoring configuration language, compatible with any monitoring system deployed via the Omnia monitoring factory.

In conclusion, comparing the proposed research solution with challenges and pitfalls observed in industrial practice (see Section 8), we argue that although in a prototype stage, *Omnia* and connected technical contributions offer a valuable basis to enter the complex and often (very) expensive world of monitoring infrastructures for cloud applications.

9.1 Research Playground

This section outlines the organizational and socio-technical scenario that Omnia was designed to address. More in particular, we elaborate on the domain assumptions typical of the scenario we have in mind. Even though the approach could be extended to different usage scenarios, for the design of an initial prototype we consider a scenario in which a cloud application is structured according to the microservices architecture pattern along with

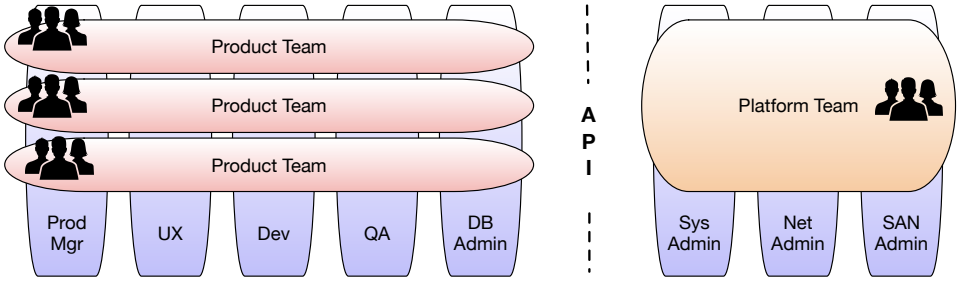

Figure 9.1: *Team organization at Netflix. Retailored from [42]*

the typical organizational-social structure [50] connected to that pattern - the scenario we address is tailored from the one adopted at Netflix [42] (Figure 9.1).

9.1.1 Domain Assumptions

Omnia assumes that each *product team* is responsible for its own product (or service), which is implemented as a microservice, for which source-code is maintained in a separate versioned repository and following an organization where development and deployment cycles are still independent from each other. Conversely, the *platform team* is cross-functional: it is in charge of supporting product teams providing infrastructure support, e.g., via APIs, orchestration software, middleware and similar technology. Such platforms are either managed by a public cloud provider, managed in-house, or a mix of these two. However, according to Netflix, there are at least three key properties that shall define the platform usage: "API-driven, self-service and automatable" [42].

9.1.2 Motivations

Standardizing a way to describe what every product team would like to see and be notified about is definitely challenging, since every monitoring tool has its own peculiarity and is usually focused on delivering value from a specific perspective. For example, a graphing tool may be able to plot multiple time series on the same graph for simplifying the comparison, or some analysis tool may be able to compute prediction or perform statistics that another tool is not able to perform. Or else, some tool may be able to send app notifications while another is only able to send emails.

With Omnia, our goal is to find a reasonable subset of standard features a company with small to medium cloud resources (*e.g.*, personnel, exper-

tise, consultancy, budget or otherwise) would like to have available, from a monitoring perspective. Omnia assumes that, stemming from these standard features, that very same company can gradually and incrementally: (a) add new features to its own indoor monitoring “language"; (b) push monitoring tool vendors to implement the missing ones or alternatively, (c) elaborate further on their own monitoring (micro)services to fulfill new feature requests.

In this scenario, every product team can describe its monitoring configuration using a versionable configuration language, which we call *monitoring-configuration-as-code*, and keep that code versioned together with its services code in the root of its repository. The next section elaborates further on this key idea, which constitutes the basis of the Omnia approach.

9.2 The Omnia approach

In this section we provide an overview of Omnia to showcase its benefits, more details are provided in the following sections.

Considering an organizational and social structure like the one we described in Section 9.1.1, we describe how monitoring and its management is addressed today and how it can be addressed with our approach.

Figure 9.2 depicts a fictional scenario where a company with low budget constraints wants to monitor its microservices architecture using existing open source monitoring tools. After studying existing solutions, the platform team decides to build a monitoring system composed of 3 different components on the server side: the temporal database *TDB X* for storing historical data, the *Dashboard X* for exposing time-series in form of graphs via a web interface, and *Alerting X* for sending notifications to product teams. Moreover, the team decides to adopt *Agent X* as data collectors to be run as daemons on the hosts.

On the left of Figure 9.2 the reader can look at how the adoption would work according to the classical approach. The platform team has to learn how to use the different tools, configure and deploy them. The platform team would then ask product teams to instrument their microservice with a vendor dependent instrumentation library, *i.e.*, the *TDB X Lib*. Product teams have to learn how to configure and use the graphical user interfaces provided by the deployed monitoring tools in order to setup their graphs and alerts. The entire process requires a steep learning curve and most of the work is manual or can be automated using custom scripts.

By using Omnia, on the other hand, most of the process is automated. The platform team has to describe the system using the proposed infras-

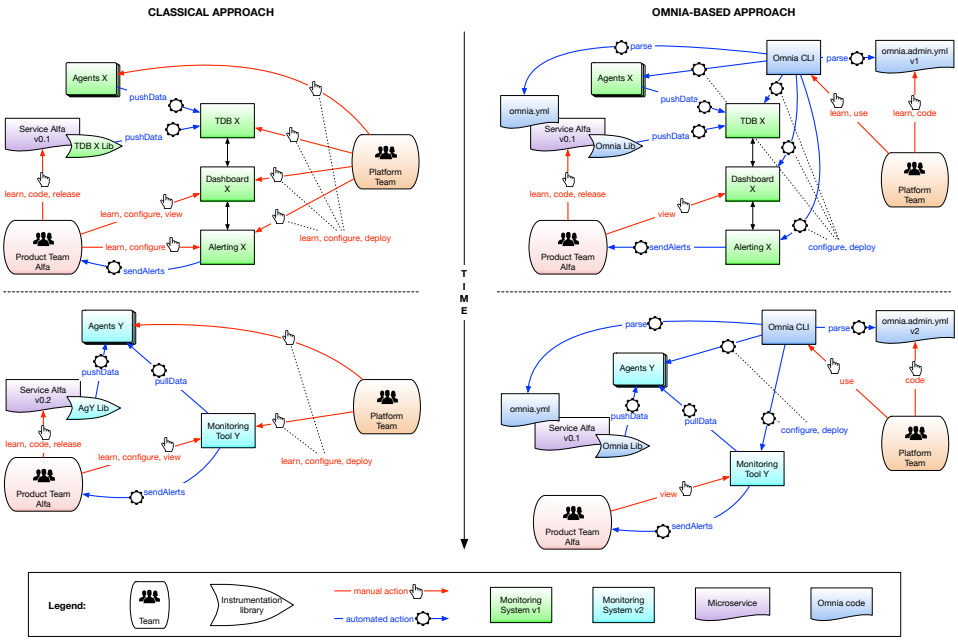

Figure 9.2: *Comparison between the classical approach and the Omnia approach when adopting different monitoring solutions.*

tructure-as-code approach. The automated setup and deployment is carried out transparently by the *Omnia Command Line Interface* (CLI), using a convention-over-configuration approach. Product teams describe their graphs and alerts using the proposed configuration-as-code approach and keep the file versioned in their code base.

After few months of practical experience with the installed monitoring system, performance problems as well as usability issues are raised and the platform team decides to switch to a more simple all-in-one monitoring solution, offering storing, graphing and alerting features in a single application and using a pull strategy for retrieving data instead of having agents pushing data to the time series database (bottom side of Figure 9.2). On the left side, we can see how the migration process would work in a classical scenario. Most of the effort carried out by all teams is thrown away, and an additional learning step is required. The platform team has to configure the new platform and deploy it. Product teams have to release a new version of the microservice with a new instrumentation library, *i.e.*, *AgY Lib*, has to learn how to use the new graphical interface of *Monitoring Tool Y* and reconfigure all required graphs and alerts.

By using Omnia, on the other hand, no additional learning step is re-

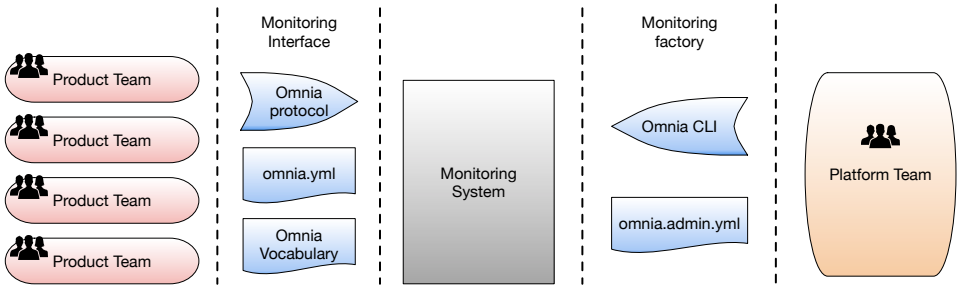

Figure 9.3: *Omnia technological contributions, depicted in blue*

quired. Product teams are not even required to touch their code. They just start using the new dashboard, with the same kind of graphs they defined for the first version of the platform already available. Alerts will be received as well as configured in the previously released configuration-as-code file. The platform team only requires to update the infrastructure-as-code file and trigger a new deployment phase via the *Omnia CLI*.

Both the product- and the platform-team workflows can be reiterated multiple times independently.

In the following sections we are going to detail the technological contributions we overviewed in the above example scenario. Figure 9.3 depicts such contributions and highlights how such decoupling between teams is obtained.

9.2.1 The monitoring interface

The Omnia vocabulary

The *Omnia vocabulary* is a dictionary of terms defining naming conventions for resource types and metrics that are common to all applications. It is supposed to be extended and maintained together with Omnia development and the addition of tools and libraries. Whenever Omnia is extended for supporting a new collecting tool, a mapping between such tool vocabulary and the Omnia one should be found and a translation implemented. If some metric or resource is missing, this should be added to the Omnia vocabulary. The same is valid when building instrumentation libraries. Users can obviously specify custom metrics, but meta-data such as the application name should be added to monitoring data, possibly in a transparent way, using terms from the Omnia vocabulary.

Here is a first version of the Omnia vocabulary with some examples of resources definitions:

Resource	Description
host	a physical or virtual machine
service	an application
service_id	a unique identifier for an instance of an application
container_id	a unique identifier for a Linux container
container_image	a Linux container image

Metrics can be categorized according to the resource being monitored. For example, here is a short list for host and Java metrics:

Host metrics	Java metrics
cpu_usage_user	*heap_memory_usage*
cpu_usage_system	*thread_count*
cpu_usage_idle	*loaded_class_count*
mem_used	*garbage_collection_time*
mem_used_percent	*thread_count*

The Omnia protocol and instrumentation libraries

The Omnia protocol specifies how monitoring data should be serialized and sent by data collectors to the other tools composing the monitoring system. We decided to adopt an existent and widely adopted protocol, that supported multidimensional meta-data in form of key values, since we could rely on existing community provided libraries and ease the adoption of industries. Such protocol is the Statsd protocol, with Influxdb tagging extension[1].

Once a protocol is defined, it is important to maintain instrumentation libraries that adhere to such protocol and enforce Omnia conventions. There exist an ever growing number of languages and frameworks, and each combination of these requires a library. A Omnia-compatible instrumentation library must adhere to the following mandatory requirements:

- *MR1.* Use the Omnia protocol, *i.e.*, the Influx Statsd protocol to serialize metrics and send metrics;
- *MR2.* Use the common Omnia vocabulary (Section 9.2.1) for decorating metrics with meta-data;
- *MR3.* Set *http://collector:8125* as default endpoint for sending metrics.

[1] `https://www.influxdata.com/getting-started-with-sending-statsd-metrics-to-telegraf-influxdb/`

Also it should adhere to the following optional requirements:

- *OR1.* Require the least possible instrumentation effort and overhead to product teams;
- *OR2.* Offer an API that is not supposed to change in the near future;
- *OR3.* Favor convention over configuration, for example by automatically inferring meta-data to be added to monitoring data.

For a first prototype, a Java library for the Spring Cloud framework[2] was developed[3]. Instrumentation only requires to add a Maven[4] dependency to the project and decorate the main application class with the *@EnableOmnia* annotation. This would automatically enables the collection of default Spring Boot Actuator metrics[5] (*e.g.*, heap memory usage or thread count) to the default endpoint (*i.e.*, *http://collector:8125*) and the addition to all metrics of the *service* and *service_id* meta-data. Moreover, developers can easily describe additional custom metrics, such as the number of payments processed by the service instance, using the API provided by the Spring Boot Actuator library as described in the following example.

```
@Service
public class MyService {
    private final CounterService counter;

    @Autowired
    public MyService(CounterService counter) {
        this.counter = counter;
    }

    public void pay() {
        this.counter.increment("payments");
    }
}
```

Listing 9.1: *Custom metrics instrumentation*

In the prototype developed for this work, additional system level meta-data such as the host name where the service is running, will be added to each metric by the data collector (or agent).

Monitoring-configuration-as-code

The *omnia.yml* file is a versionable YAML file used for configuring runtime monitoring, such as metrics time series to be plotted or alerts.

[2] http://projects.spring.io/spring-cloud/
[3] https://github.com/mmiglier/omnia-spring-boot
[4] https://maven.apache.org
[5] https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html

The first version of the *omnia.yml* is composed of two sections: (1) the *dashboard* section, where things to be visualized are described, and (2) the *action* section, where things to be done in response to events are configured. The dashboard can be composed of different kind of graphs, such as time series or pie charts. In the action section, product teams can describe actions such as email or SMS notifications, but could also adaptive actions to be triggered.

An example of *omnia.yml* file is shown in the following listing:

```
dashboard:
  timeseries:
    - metric: payments
      compute: rate
    - metric: java_heap_memory
    - metric: cpu
      compute: average by host
    - metric: ram
actions:
  email:
    - condition: http_errors / http_requests > 0.1
```

Listing 9.2: *omnia.yml file for monitoring configuration.*

The file shall be automatically validated during automatic integration tests. Also, the file shall favor convention over configuration: every missing piece of information shall be configured using standardizable defaults.

Whenever a new version of the *omnia.yml* file is pushed, Omnia will update the product team specific dashboard together with any alert specified in the document. If this configuration file was standard and shared among tools, the monitoring team would be free to update the tool set of monitoring tools without interfering with the product teams work. During a first phase, interpreters and translators should be provided to compile the standard configuration file into the tool specific configuration format, with the hope an increase of popularity of the standard format, it may become widely adopted.

9.2.2 The monitoring factory

Monitoring-infrastructure-as-code

The *omnia.admin.yml* file depicted in Figure 9.3 is a file where the monitoring system is described as infrastructure-as-code and product teams repositories are listed. The first prototype of this file was inspired by the *docker-compose.yml* files used by Docker[6] to describe containers architectures, from which we took the simplistic approach.

[6] www.docker.com

This configuration file is composed of 3 parts (1) the *provisioner*, where the platform team is supposed to specify the provisioner the *Omnia CLI* should use to provision the monitoring system, (2) the *tools* section, where the platform team has to list the monitoring tools, the functionalities they offer and their interconnections, and (3) the *team_repos* section, where product teams repositories are listed.

We here provide an example of *omnia.admin.yml* file:

```
provisioner:
  name: docker
  args:
    username: mmiglier
    images_tag: latest
tools:
  telegraf:
    provides:
        - agent
    pushes_to:
      - influxdb
  influxdb:
  grafana:
    pulls_from:
      - influxdb
    provides:
        - dashboard
        - actions
teams_repos:
  - "github.com/mmiglier/omnia-examples/service1"
  - "github.com/mmiglier/omnia-examples/service2"
  - "github.com/mmiglier/omnia-examples/service3"
```

Listing 9.3: *An example of omnia.admin.yml file.*

The Omnia CLI

The *Omnia CLI* is the actual *monitoring factory*, the application that is used by the platform team to deploy a monitoring system that implements the *monitoring interface* according to the *omnia.admin.yml* (see Figure 9.3). The application exposes three simple commands:

- ***compile***, the CLI parses the *omnia.admin.yml* file, retrieves *omnia.yml* files from team repositories and creates the required configuration files required by the chosen provisioner to deploy the monitoring system;
- ***deploy***: the CLI deploys the monitoring system using the API offered by the chosen provisioner;
- ***stop***: the CLI stops the monitoring system using the API offered by the chosen provisioner.

The platform is written in Go[7] and is easily extensible with new provisioners and new monitoring tools by using the Go *template package*[8]. During compilation time, both provisioner's and tools' configuration files are generated from templates by applying to them a data structure generated from the *omnia.admin.yml* and the *omnia.yml* files. Besides templates for configuration, a developer extending *Omnia CLI* with a new tool has to create a *setup.sh* and a *run.sh* script, which will be executed for setting up the tool and running it respectively.

Starting from the monitoring-infrastructure-as-code example in Listing 9.3, the *Omnia CLI* will generate the following *docker-compose.yml* file after compilation:

```
version: '2'
networks:
  default:
    external:
      name: monitoring
services:
  collector:
    image: telegraf:1.1
    ports:
      - "8092:8092/udp"
      - "8094:8094"
      - "8125:8125/udp"
    volumes:
      - "./telegraf/etc:/etc/telegraf:ro"
      - "/:/rootfs:ro"
      - "/sys:/rootfs/sys:ro"
      - "/proc:/rootfs/proc:ro"
      - "/etc:/rootfs/etc:ro"
      - "/var/run/docker.sock:/var/run/docker.sock:ro"
    environment:
      - HOST_MOUNT_PREFIX=/rootfs
      - HOST_PROC=/rootfs/proc
      - HOST_SYS=/rootfs/sys
      - HOST_ETC=/rootfs/etc
    entrypoint: /bin/sh
    command: -c "sleep 10s && telegraf"
  influxdb:
    image: influxdb:1.1
    ports:
      - "8086:8086"
  grafana:
    image: grafana/grafana:4.0.2
    ports:
      - "80:3000"
    volumes:
      - "./grafana/etc:/etc/grafana:ro"
      - "./grafana/run.sh:/run.sh:ro"
```

[7] https://golang.org
[8] https://golang.org/pkg/text/template/

```
    cmd: "/run.sh"
```

Listing 9.4: *docker-compose.yml file generated from the omnia.admin.yml file in Listing 9.3*

A prototype implementation of this component has been released on GitHub[9].

9.3 Discussion and Future Work

The approach proposed in this chapter is still lacking an extensive evaluation. However, we believe it addresses the challenges identified in the research study presented in Chapter 8. The lack of standards is faced by introducing a common API and a simple DSL for building and configuring monitoring infrastructures using existing open source monitoring tools. The proposed approach also decouples development teams since they are not bound to a specific monitoring infrastructure, they use a the Omnia interface for sending data and the Omnia configuration language to declare what they need. Such decoupling helps operators as well, since they can more easily experiment with new monitoring tools without having developers to change their repositories.

The work has been presented in a paper submitted to the 3rd International Workshop on Quality-Aware DevOps (QUDOS 2017)[10] where we hope to receive a first feedback.

As future work it is our intention to create and foster the growth of an open-source community around the concept introduced as part of this chapter, empirically evaluating the approach by looking at the feedback we may receive from practitioners.

[9] https://github.com/mmiglier/omnia
[10] http://qudos2017.fortiss.org

CHAPTER 10

Conclusion

This thesis focused on monitoring modern distributed applications, investigating challenges for both monitoring tools developers and users, and proposing possible solutions. We claimed that monitoring is fundamental for having a fast feedback from production, as a continuation of verification and validation in the runtime.

Tower 4Clouds is an approach and a platform we proposed to solve the identified challenges listed in Section 1.1. The following requirements have been addressed:

- *Abstract from heterogeneity and prevent lock-in* (Requirement 1.1.1). It was addressed by proposing a model based approach for describing the software system, QoS constraints and monitoring rules in a provider independent way and investing different abstraction layers.
- *Elastically adapt to ephemeral and dynamic systems* (Requirement 1.1.2). It was addressed by implementing an autonomous discovery mechanism where data collectors actively inform the server of their existence and update the runtime model of the system.
- *Limit the requirements on the data collector side to improve portability* (Requirement 1.1.3). It was addressed by defining a monodirec-

tional protocol specification for data collectors which does not asks for requirements such as port binding on the client side.

- *Provide an extensible platform able to cope with future evolutions and interoperate with existing tools* (Requirement 1.1.4). It was addressed by offering a flexible and extensible framework which allowed to integrate existing monitoring tools and implement additional actions.
- *Timely provide required information for reacting before end-user perception* (Requirement 1.1.5). It was partially addressed by allowing the user to configure the evaluation frequency of monitoring rules.

Next, an industrial survey was run with the objective of understanding what challenges monitoring tools users are perceiving. The study was conducted by first personally interviewing 7 practitioners. This allowed us to have a first perception of the state of practice and guided us to the formulation of an online form which was answered by 141 practitioners around the world. The survey allowed us to answer to the following questions:

- *Q1. Is monitoring actually perceived as a fundamental asset?* The high participation to the study revealed a widespread interest in monitoring. Among the surveyed companies only 13% does not consider monitoring worth the investment. However, these very companies result to have lower availability.
- *Q2. Do all companies monitor their software systems and how?* Even though monitoring is considered to be important, its setup is often immature: several companies only perform manual checks via terminal (18%), 36% does not use any third party monitoring tool, they develop their own scripts and dashboard. Finally, among the ones that use third party monitoring tools, there is no *de facto* standard, every company seems to create its personal composition of tools.
- *Q3. How are incidents discovered and handled?* Incidents are discovered mainly via automatic emails, by manual inspecting logs or graphs or directly from customers. Most of the problem are solved manually, while automatic remediation is mainly based on scaling up and down machine, based on CPU and memory thresholds, or by restarting services.
- *Q4. What are the people involved with monitoring?* We discovered that it is quite common today to have developers and even the business to receive alerts about incidents.

- *Q5. What are the most critical challenges perceived when trying to make a system observable?* The proliferation of tools without standards defined and the lack of usability are the most critical challenges perceived by monitoring tools users.
- *Q6. Does Tower 4Clouds provide any contribution that may help companies?* Tower 4Clouds provides a contribution by creating an extensible framework and proposing hints to a possible way of standardization.

Open challenges identified from the survey are the following:

- *C1. Lack of standards.* There is no standard protocol for sending monitoring data and no standard schema for declaring them.
- *C2. Strong dependencies among teams.* Monitoring is a support service, required by all other services composing the system. Changes to the monitoring platform affect all other teams focusing on those services.
- *C3. Fast evolution and tool proliferation*. There are new technologies and new environments coming out every other day, together with a new monitoring tool.
- *C4. Lack of reusability*. Each company creating its preferred monitoring infrastructure is implementing its own automation scripts for deploying and configuring it.

These challenges are, to our knowledge, currently unsolved. In Chapter 9 we have presented the Omnia approach that addresses, at least partially, these challenges.

Omnia is an approach and a tool with the key objective of reducing the learning curve and entry-cost to monitoring technologies. The concept of *monitoring factory* was introduced, as a reinterpretation of the famous design pattern where the concrete implementation of a monitoring system is kept hidden to developers via a common monitoring interface. Deployment and configuration of the monitoring platform is automated via the simple API offered by the monitoring factory we presented. System administrators can leverage the proposed monitoring-infrastructure-as-code to easily compose the set of existing monitoring tools to use and configure their roles and interconnections.

Our monitoring interface allows to separate the development workflow of the core application from the monitoring system, increasing agility and

reduce effort required. The monitoring factory permits to switch across different monitoring solutions, automating the deployment of a solution that is compliant with our monitoring interface.

The approach is mainly ongoing work and is still lacking an extensive. As future work it is our intention to create and foster the growth of an open-source community around the concept introduced, empirically evaluating the approach by looking at the feedback we may receive from practitioners.

Bibliography

[1] Apache kafka. http://kafka.apache.org.

[2] Chef. https://www.chef.io.

[3] Collectd. https://collectd.org.

[4] Consul. https://www.consul.io.

[5] Grafana. http://grafana.org.

[6] Graphite. http://graphite.readthedocs.org.

[7] Nagios. https://www.nagios.org.

[8] Oxford dictionaries. https://www.oxforddictionaries.com.

[9] Prometheus. https://prometheus.io.

[10] Puppet. https://puppetlabs.com.

[11] Rabbitmq. http://www.rabbitmq.com.

[12] Redis. http://redis.io.

[13] Riemann. http://riemann.io.

[14] Sensu. https://sensuapp.org.

[15] Statsd. https://github.com/etsy/statsd.

[16] Swrl: A semantic web rule language combining owl and ruleml. http://www.w3.org/Submission/SWRL/.

[17] Zookeeper. https://zookeeper.apache.org.

[18] Announcing amazon elastic compute cloud (amazon ec2) - beta. https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/, 2006 (accessed September 6, 2016).

[19] Docker: Automated and consistent software deployments. https://www.infoq.com/news/2013/03/Docker, 2013 (accessed September 19, 2016).

[20] Release: Aws lambda on 2014-11-13. https://aws.amazon.com/releasenotes/AWS-Lambda/8269001345899110, 2014 (accessed September 19, 2016).

[21] Serverless architectures. `http://www.martinfowler.com/articles/serverless.html`, 2016 (accessed September 19, 2016).

[22] Giuseppe Aceto, Alessio Botta, Walter de Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093 – 2115, 2013.

[23] Danilo Ardagna, Michele Ciavotta, Giovanni Paolo Gibilisco, Riccardo Benito Desantis, Giuliano Casale, Juan F. Pérez, Francesco D'Andria, and Román Sosa Gonzalez. Qos assessment and sla management. In Elisabetta Di Nitto, Peter Matthews, Dana Petcu, and Arnor Solberg, editors, *Model-Driven Development and Operation of Multi-Cloud Applications*, chapter 4, pages 31–41. Springer International Publishing, 2016.

[24] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying rdf streams with c-sparql. *SIGMOD Rec.*, 39(1):20–26, September 2010.

[25] Luciano Baresi and Sam Guinea. Event-based multi-level service monitoring. In *2013 IEEE 20th International Conference on Web Services*, pages 83–90, June 2013.

[26] Joshua Barratt. Getting more signal from your noise. `http://serialized.net/2011/02/getting-more-signal-from-your-noise/`, 2011.

[27] Antonia Bertolino. Software testing and/or software monitoring: Differences and commonalities. Jornadas Sistedes, 2014.

[28] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE software*, 1(1):75, 1984.

[29] Lorenzo Cianciaruso, Francesco di Forenza, Elisabetta Di Nitto, Marco Miglierina, Nicolas Ferry, and Arnor Solberg. Using models at runtime to support adaptable monitoring of multi-clouds applications. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2014 16th International Symposium on*, pages 401–408. IEEE, 2014.

[30] Adrian Cockroft. Monitoring challenges. Monitorama, 2016.

[31] Rustem Dautov, Iraklis Paraskakis, and Mike Stannett. Utilising stream reasoning techniques to underpin an autonomous framework for cloud application platforms. *Journal of Cloud Computing*, 3(1), 2014.

[32] Emanuele Della Valle, Stefano Ceri, Davide Francesco Barbieri, Daniele Braga, and Alessandro Campi. *A First Step Towards Stream Reasoning*, pages 72–81. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[33] Elisabetta Di Nitto, Peter Matthews, Dana Petcu, and Arnor Solberg. *Model-Driven Development and Operation of Multi-Cloud Applications*. Springer International Publishing, 2017.

[34] Frank Elberzhager, Jürgen Münch, and Vi Tran Ngoc Nha. A systematic mapping study on the combination of static and dynamic quality assurance techniques. *Inf. Softw. Technol.*, 54(1):1–15, January 2012.

[35] Kaniz Fatema, Vincent C. Emeakaroha, Philip D. Healy, John P. Morrison, and Theo Lynn. A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *Journal of Parallel and Distributed Computing*, 74(10):2918–2933, 2014.

[36] Nicolas Ferry, Marcos Almeida, and Arnor Solberg. The modaclouds model-driven development. In Elisabetta Di Nitto, Peter Matthews, Dana Petcu, and Arnor Solberg, editors, *Model-Driven Development and Operation of Multi-Cloud Applications*, chapter 3, pages 21–30. Springer International Publishing, 2016.

[37] Nicolas Ferry and Arnor Solberg. Models@runtime for continuous design and deployment. In Elisabetta Di Nitto, Peter Matthews, Dana Petcu, and Arnor Solberg, editors, *Model-Driven Development and Operation of Multi-Cloud Applications*, chapter 9, pages 77–89. Springer International Publishing, 2016.

[38] Nicolas Ferry, Arnor Solberg, Pooyan Jamshidi, Rasha Osman, Weikun Wang, Stepan Seycek, Vanessa Gligor, Roi Sucasa, and Antonin Abhervé. Modaclouds evaluation report: Final version, 2015.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object oriented software, 1995.

[40] IBM. *IBM Dictionary of Computing*. McGraw-Hill, Inc., New York, NY, USA, 10th edition, 1993.

[41] Alessandro Margara, Jacopo Urbani, Frank van Harmelen, and Henri Bal. Streaming the web: Reasoning over dynamic data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 25(0), 2014.

[42] Tony Mauro. Adopting microservices at netflix: Lessons for team and process design. `https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/`, 2015 (accessed January 16, 2017).

[43] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[44] Marco Miglierina, Marco Balduini, Narges Shahmandi Hoonejani, Elisabetta Di Nitto, and Danilo Ardagna. Exploiting stream reasoning to monitor multi-cloud applications. In Irene Celino, Emanuele Della Valle, Markus KrÃ¶tzsch, and Stefan Schlobach, editors, *OrdRing@ISWC*, volume 1059 of *CEUR Workshop Proceedings*, pages 33–36. CEUR-WS.org, 2013.

[45] Alessandro Orso. Monitoring, analysis, and testing of deployed software. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 263–268, New York, NY, USA, 2010. ACM.

[46] Greg Poirier. Monitoring is dead, long live monitoring. Monitorama, 2016.

[47] Rick Rabiser, Sam Guinea, Michael Vierhauser, Luciano Baresi, and Paul GrÃ¼nbacher. A comparison framework for runtime monitoring approaches. *Journal of Systems and Software*, 125:309 – 321, 2017.

[48] William Robinson. A roadmap for comprehensive requirements modeling. *Computer*, 43(5):64–72, May 2010.

[49] Forrest Shull, Victor R. Basili, Barry W. Boehm, A. Winsor Brown, Patricia Costa, Mikael Lindvall, Daniel Port, Ioana Rus, Roseanne Tesoriero, and Marvin V. Zelkowitz. What we have learned about fighting defects. In *IEEE METRICS*. IEEE Computer Society, 2002.

[50] Damian Andrew Tamburri, Patricia Lago, and Hans van Vliet. Organizational social structures for software engineering. *ACM Comput. Surv.*, 46(1):3, 2013.

[51] Demetris Trihinas, George Pallis, and Marios D. Dikaiakos. Jcatascopia: Monitoring elastically adaptive applications in the cloud. In *CCGRID'14*, pages 226–235, 2014.

[52] Michael Vierhauser, Rick Rabiser, and Paul Grünbacher. Requirements monitoring frameworks: A systematic review. *Information and Software Technology*, 2016.

[53] Michael Vierhauser, Rick Rabiser, Paul Grünbacher, Klaus Seyerlehner, Stefan Wallner, and Helmut Zeisel. Reminds : A flexible runtime monitoring framework for systems of systems. *Journal of Systems and Software*, 112:123 – 136, 2016.

[54] John E. Vincent. Monitoring sucks - watch your language. `http://lusislog.blogspot.it/2011/07/monitoring-sucks-watch-your-language.html`, 2011 (accessed November 14, 2016).

[55] Liming Zhu, Len Bass, and George Champlin-Scharff. Devops and its practices. *IEEE Software*, 33(3):32–34, 2016.

APPENDIX A

Ticket Monster Instrumentation

In this appendix chapter we provide a code snippet to show how the Ticket Monster Java EE example application, introduced in Section 4.2, can be instrumented using our Java data collector. It suffices to add the following class:

```
package org.jboss.jdf.example.ticketmonster.monitoring;

import it.polimi.tower4clouds.java_app_dc.Property;
import it.polimi.tower4clouds.java_app_dc.Registry;

import java.util.HashMap;
import java.util.Map;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

import org.jboss.jdf.example.ticketmonster.rest.BaseEntityService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@WebListener
public class ConfigurationInitializer implements ServletContextListener {

        @Override
        public void contextDestroyed(ServletContextEvent sce) {
                Registry.stopMonitoring();
        }
```

```
    @Override
    public void contextInitialized(ServletContextEvent sce) {
            Map<Property, String> applicationProperties = new HashMap<
                Property, String>();
            applicationProperties.put(Property.ID, loadVariable("
                MODACLOUDS_TOWER4CLOUDS_INTERNAL_SERVICE_ID", "_ic1"))
                ;
            applicationProperties.put(Property.TYPE, loadVariable("
                MODACLOUDS_TOWER4CLOUDS_INTERNAL_SERVICE_TYPE","WebApp
                "));
            applicationProperties.put(Property.EXTERNAL_VM_ID,
                loadVariable("MODACLOUDS_TOWER4CLOUDS_EXTERNAL_VM_ID",
                 "_vm1"));
            applicationProperties.put(Property.EXTERNAL_VM_TYPE,
                loadVariable("MODACLOUDS_TOWER4CLOUDS_EXTERNAL_VM_TYPE
                ","WebVM"));
            applicationProperties.put(Property.VENDOR_ID, loadVariable
                ("MODACLOUDS_TOWER4CLOUDS_VENDOR_ID", "_cp1"));
            String mpIP = loadVariable("
                MODACLOUDS_TOWER4CLOUDS_MANAGER_IP", "localhost");
            String mpPort = loadVariable("
                MODACLOUDS_TOWER4CLOUDS_MANAGER_PORT", "8170");
            Registry.initialize(mpIP, Integer.parseInt(mpPort),
                applicationProperties,
                        BaseEntityService.class.getPackage().
                            getName(), false, true);
            Registry.startMonitoring();
    }

    public static String loadVariable(String variableName, String
        defaultValue) {
            if (System.getenv().containsKey(variableName)) {
                    return System.getenv(variableName);
            }
            return defaultValue;
    }

}
```

Listing A.1: *Ticket Monster instrumentation code*

The code in such class is mainly composed of configuration retrieval from the environment regarding the monitored resource, the Tower 4Clouds manager endpoint and finally the code required to specify the package where annotated classes can be found. At runtime, each method with any of the JAX-RS annotations (*i.e.*, *GET*, *POST*, *PUT* and *DELETE* endpoints) will have its duration measured and collected.

APPENDIX B

Industrial Survey: Additional Resources

B.1 In-person interview questions

The list of 15 open questions used for the in-person interviews follows:

1. Can you describe your business and the high level architecture of your core software system in that business?
2. What are the most important runtime metrics you monitor for your core business? Among these metrics, which are those that you use to assess the reliability of your system, if any?
3. Do you have any issue-detection mechanism, e.g., alarm-raising based on some metrics-threshold? Can you provide incident examples? Who is notified in these cases and how?
4. Assume there is a problem (*e.g.*: some alarm goes off), what is the incident-handling procedure that you follow to react?
5. Do you have any automatic adaptation based on monitoring data? Which adaptation mechanism requires human intervention instead?
6. Is there anything that could be automated which is now performed manually?
7. Are developers kept in the operations loop and accounted for the functioning of what they built or monitoring is just an operations business?
8. Can you describe your monitoring infrastructure?
9. How do you instrument code for collecting application level metrics?
10. How do you instrument machines for collecting host resources?
11. Where are monitoring and reliability checks executed (e.g., client vs. server side)?

12. How is monitoring data transferred from data collectors to the monitoring server?
13. Are your monitoring data stored for supporting historical analyses? If yes, how is your historical monitoring data stored?
14. If you maintain historical monitoring data, how do you use it?
15. Do you see any way of improvement of the monitoring activity which you cannot find in existing monitoring tools?

B.2 In-person interview answers

An anonymized summary of the seven in-person interviews follows:

1. **I1.** The first respondent is the CTO of a medium traveling company. The company has a small IT department of around 10 people working as a unique development and operation team. They develop and maintain the platform required for selling travels for their own company. The development of their latest products was exploiting recent technologies like Amazon EC2 for the hosting, docker for containerization of their services and micro-service as architectural style. Monitoring was not perceived as a core activity since they claimed to be in a yet immature development phase. Monitoring for them consisted merely on default metrics offered by their cloud provider (*i.e.*: Amazon Cloud Watch) and availability checks. Incidents are mainly resolved looking at logs stored on machines. In the short time they are mainly interested in business metrics to understand better what users expect from them and what problems they encounter in terms of content satisfaction.
2. **I2.** The second interviewee is a research software engineer working in the R&D department of a medium company, where only initial prototypes are developed and therefore they are interested only on a premature analysis of the monitoring activity in case the product becomes a commercial product. The ongoing project consisted in developing a live data analysis tool able to process news from multiple sources at high rate. The only monitoring they had in place was a custom sensor measuring the queues lengths for automatic remediation to prevent overloading and ensure results timeliness. Initial monitoring requirements for a production ready version would be metrics for guaranteeing high availability and fault tolerance. This would require automatic remediation such as auto scaling, load balancing and automatic restart and therefore data collectors for measuring required metrics (*e.g.*: availability, latency, memory utilization).
3. **I3.** The third interviewee is a system architect in a medium technological company offering SaaS services for enterprises business management. They deploy their product on top of both Windows and Linux virtual machines by means of Puppet scripts. Their monitoring is mainly focused on database related metrics since their SQL servers are the critical part for their business. Important metrics are, therefore, related to disk, caching, requests and processes on these servers, besides the classical CPU and memory utilization. Numerical data is stored on Graphite and shown on Grafana dashboards. All metrics, including binary checks, are managed via Nagios and Icinga. They have some automatic remediation which usually involves trying to restart the machine before emailing operators. Logs are managed by means of the ELK stack. Developers do not receive alerts, though they work side by side with operators when required. A feature they claim to miss in their monitoring stack is the ability of combining historical data storage and graphing in one system easily. They currently use different collecting technique on the same machines for different technologies and scopes, some agents are collecting the same metrics and sending them in multiple ways instead of using a unified approach.

4. **I4.** The fourth interviewee works in a Small enterprise composed of 5 employees, they offer a SaaS solution over Google PaaS solution. Besides availability, their main monitoring concern is about business metrics, monitored via Google Analytics. They claim they currently do not need any further metrics since most of the QoS is guaranteed by the Google platform. Performance metrics such as latency is offered by the provider and is occasionally monitored to understand if there is some query optimization required.
5. **I5.** The fifth interviewee works in a large technological company selling both hardware and required middleware to customers for running their business. They provide monitoring out of the box. He considers the most important runtime metrics to be machine memory, memory profile, paging, memory leaks checks, CPU activity, disks activity, throughput and latency. At the application level is important to monitor JVM metrics in case of Java applications, heap size, transactions and http requests. Their tools offer configurable alerting tools based on threshold. They also offer some advanced AI prediction algorithm trained on historical data. They provide some automatic remediation based such as automatic archiving when file system is full, or automatic switch to backup services in case of availability issues. According to him future improvements will be in the AI field for prediction and diagnosis.
6. **I6.** The sixth interviewee is working in another large technological company selling their products either on premise or as a service. In his experience with several customers he saw that financial companies are the ones that spend the most for monitoring since unavailability issues can cause huge loss of money. Therefore they buy very is reflected in how these companies usually address monitoring. They usually prefer monitoring as a service solutions, with small setup efforts required. Once complexity arise, then companies tend to invest in more sophisticated solutions. They also observed that most of the comp anies write custom code for monitoring, they store data in excel or relational database, based on their skills, and analyse data on demand. The interviewee considers the following metrics to be usually the most important ones for their customers: page response time, page requests, number of page requests, number of page responses. Issue detection is mostly based on thresholds and alarms are sent via email or SMS, according to problem severity. They also offer machine learning tools for recognizing common patterns. The most common alerts refer to lack of disk space. In fact, one of their major investments is on capacity planning. Other common alerts are caused by CPU or memory saturation. Unpredicted bugs on new releases often cause loops or memory leaks. Automatic remediation is often performed by restarting the machine or the service, almost nobody does automatic rollback to previous or safe versions because it may cause problems with data bases schema. Internally they are experimenting with advanced machine learning tools that are able to directly contact the responsible team and is showing to improve the MTTR (Mean Time To Repair) by 40 times. The company sees AI and automation as one of the most impacting developments in the near future for monitoring and incidents handling.
7. **I7.** The last interviewee is the CTO of a small startup with less than 50 employee offering financial services via mobile applications. Their backend is deployed on Amazon Web Services. They currently count 26 thousands users with an average of 500 thousands requests per day. The most important metrics are availability and the number of malformed requests which is monitored for security purposes. Errors are checked by manually looking at logs on the machines even though they plan soon to move to the ELK stack. Logs are automatically backed up on Amazon S3. The worst accident they experienced was caused by an expired certificate which back then was manually maintained by an employee who had left the company. We monitor CPU using Amazon Cloud Watch and we have a threshold for scaling up but it never happened so far.