PaaSage

Model Based Cloud Platform Upperware

Deliverables D5.1.2

Product Executionware

Version: 1.0

D5.1.2

Name, title and organisation of the scientific representative of the project's coordinator:

Mr Philippe Rohou Tel: +33 (0)4 97 15 53 06 Fax: +33 (0)4 92 387822 E-mail: philippe.rohou@ercim.eu

Project website address: http://www.paasage.eu

Project	
Grant Agreement number	317715
Project acronym:	PaaSage
Project title:	Model Based Cloud Platform Upperware
Funding Scheme:	Integrated Project
Date of latest version of Annex I against which the assessment will be made:	03th July 2014
Document	
Period covered:	
Deliverable number:	D5.1.2
Deliverable title	Product Executionware/
Contractual Date of Delivery:	30th September 2015 (M36)
Actual Date of Delivery:	30st September 2015
Editor (s):	Jörg Domaschka (UULM)
Author (s):	Dennis Hoppe (USTUTT), Kyriakos Kritikos (FORTH), Craig Sheridan (FLEX), Edwin Yaqub (GWDG), Jörg Domaschka (UULM), Daniel Baur (UULM), Frank Griesinger (UULM), Daniel Seybold (UULM), Bartosz Balis (AGH), Dariusz Król (AGH), Maciej Malawski (AGH), Ahmed Zarioh (be.wan)
Reviewer (s):	Kamil Figiela (AGH), Kyriakos Kritikos (FORTH)
Participant(s):	Same as authors
Work package no.:	5
Work package title:	Executionware
Work package leader:	Jörg Domaschka (UULM)
Distribution:	PU
Version/Revision:	1.0
Draft/Final:	Final
Total number of pages (including cover):	62

DISCLAIMER

This document contains description of the PaaSage project work and findings.

The authors of this document have taken any available measure in order for its content to be accurate, consistent and lawful. However, neither the project consortium as a whole nor the individual partners that implicitly or explicitly participated in the creation and publication of this document hold any responsibility for actions that might occur as a result of using its content.

This publication has been produced with the assistance of the European Union. The content of this publication is the sole responsibility of the PaaSage consortium and can in no way be taken to reflect the views of the European Union.

The European Union is established in accordance with the Treaty on European Union (Maastricht). There are currently 28 Member States of the Union. It is based on the European Communities and the member states cooperation in the fields of Common Foreign and Security Policy and Justice and Home Affairs. The five main institutions of the European Union are the European Parliament, the Council of Ministers, the European Commission, the Court of Justice and the Court of Auditors. (http://europa.eu)

PaaSage is a project funded in part by the European Union.

Executive Summary

The Executionware constitutes a fundamental part of the entire PaaSage system and its architecture. The primary purposes of the Executionware are *(i)* to enact interacting with the cloud providers through their respective and largely inhomogeneous APIs, in order to support the creation, configuration as well as tear down of virtual machines and virtual networks; *(ii)* to enact the deployment of application components such as load balancers, databases, and application servers across the created virtual machines; *(iii)* to support the monitoring of both virtual machines and application component instances by provisioning of appropriate sensors/probes and by supporting a reporting interface for applications; *(iv)* to support the aggregation of raw metrics coming from the probes to higher-level composite metrics, and the evaluation of any of these metrics according to conditions and thresholds; *(v)* to report back metric values to the Upperware and to store them in the Metadata-Database (MDDB).

Throughout the PaaSage project, the Executionware has been designed and developed in order to fulfill these tasks. Following the concept of *divide et impera* this has led to a set of components that all support a sub set of the requested features, but whose interplay emerges to the desired functionality. Most components are collected in the CLOUDIATOR suite: COLOSSEUM runs in the domain of the PaaSage operator and represents the central access point for any clients through a REST API as well as a Web UI. SWORD is a library that provides an abstraction layer for the various cloud providers. In particular, it encapsulates the differences between them with respect to terminology and technology. LANCE runs in the cloud domain. In particular, COLOSSEUM will deploy one instance of LANCE on each virtual machine it creates. LANCE is responsible for executing the life-cycle of the component instances to be installed on the virtual machine. Just as LANCE, VISOR runs in the cloud domain and is responsible for collecting monitoring data from virtual machines and component instances. In particular, COLOSSEUM will install an instance of VISOR in any virtual machine it creates. Whenever COLOSSEUM is requested to monitor certain aspects of an application and/or virtual machine, it will connect to the VISOR instance running on that virtual machine and request the installation of a sensor together with an interval.

AXE is a two-purpose component that runs partially in the home and partially in the cloud domain. Its first task is to post-process the monitoring data collected by the visor component. In particular, AXE is capable of executing aggregation functions on the monitored data such as computing amongst other statistical functions like averages, medians, and quantiles. It may relay selected metrics to third party components including the Upperware.

The `MetricsCollector` component constitutes another multi-purpose component. The respective software runs as a daemon that can be operated either in local or global mode. In local mode, it operates in specific virtual machines and replaces parts of the functionality of CLOUDIATOR's AXE; hence, `MetricsCollector` instances running in local mode are optional and dependent on the configuration of the Executionware. In contrast, in global mode, the `MetricsCollector` component provides a publish/subscribe mechanism to interested components of the Upperware and enables registering for metric data as well as violations of metric conditions, i.e., events in the meta-data database (CDO server [3]).

Intended Audience

This deliverable is a public document intended for readers with some experience with cloud computing and cloud middleware. It presumes that the reader is familiar with the overall PaaSage architecture as described in deliverable D1.6.1 [2].

For the external reader, this deliverable provides an insight into the Executionware sub-system of PaaSage, its architecture and its various entities.

For the research and industrial partners in PaaSage, this deliverable provides an understanding of the basic design and architecture of the Executionware, its capabilities, but also its limitations.

Contents

1 Introduction and Overview **13**
- 1.1 The Executionware in PaaSage . . . 13
- 1.2 The Executionware Architecture . . . 15
  - 1.2.1 CLOUDIATOR and its Tools . . . 16
  - 1.2.2 The `MetricsCollector` Component . . . 17
  - 1.2.3 Third Party Components . . . 19
- 1.3 List of Changes from Initial Prototype . . . 19
- 1.4 Structure of This Document . . . 20

2 CLOUDIATOR **21**
- 2.1 COLOSSEUM . . . 21
  - 2.1.1 Registries . . . 22
  - 2.1.2 Usage and APIs . . . 22
  - 2.1.3 Further Features . . . 26
- 2.2 SWORD . . . 27
- 2.3 LANCE . . . 27

3 Monitoring and Auto-scaling **29**
- 3.1 VISOR . . . 30
  - 3.1.1 Available Sensors and Probes . . . 31
  - 3.1.2 Application-specific Probes . . . 33
  - 3.1.3 Application specific probes for Scalarm and HyperFlow engines . . . 35
  - 3.1.4 Client libraries for Visor . . . 36
- 3.2 TSDB Selection . . . 36
  - 3.2.1 Requirements . . . 36
  - 3.2.2 KairosDB . . . 37
  - 3.2.3 OpenTSDB . . . 38
  - 3.2.4 InfluxDB . . . 39

3.2.5 Selection of TSDB 40
3.2.6 Time-series Database Installation 41
3.3 AXE 42
3.3.1 Aggregation 42
3.3.2 Scalability Rules Language 43
3.3.3 Auto-Scaling 44
3.4 MetricsCollector 44
3.4.1 MetricsCollector Modes of Operation 45
3.4.2 Main Assumptions 46
3.4.3 Architecture 46
3.4.4 Modes of Interaction 51
3.4.5 Integration with AXE 51

4 Further Aspects **53**
4.1 Testbeds 53
4.1.1 GWDG's OpenStack Testbed 53
4.1.2 Flexiant Testbed (FLEX) 54
4.2 Deployment Controller 57

5 Conclusion and Future Work **59**

Bibliography **61**

Terminology

Throughout this document we use a set of terms with an overloaded meaning. Therefore, this section aims at defining these terms for this document in a brief and concise manner. Throughout this deliverable, all of the terms defined here are exclusively used according to our definition and not in any other way.

Cloud Terminology

Cloud platform A cloud platform refers to a software stack and accordingly to the API offered by that stack. As the platform is something completely passive and not a concrete offer (cf. cloud) it does, however, not define contact endpoints (e.g. URIs). The OpenStack software suite and Flexiant Cloud Orchestrator (FCO) are examples of cloud platforms.

Cloud provider A cloud provider is an organizational entitiy or some other kind of actor that runs a cloud platform under a dedicated endpoint/URL (e.g. RedStack). This means, it defines the access points (i.e. URIs) to access the services offered. In addition to that, legal aspects are tied to the cloud provider. Amazon EC2 is an example of a cloud provider. Within the PaaSage consortium examples include GWDG running an OpenStack cloud platform and Flexiant running a FCO cloud platform.

Cloud A cloud refers to a cloud platform offered by a cloud provider as seen by a tenant. That is, besides the endpoint of the provider, *a cloud* (in contrast to *the Cloud*) is also linked to log-in credentials such as username and password.

Application Terminology

(Cloud) application A *(cloud) application* is a possibly distributed application consisting of one or multiple interlinked application components. As such, an application is solely a description and does not represent anything enacted.

(Application) component An *(application) component* for short is the smallest divisible element of an application. It is the unit of scale and the unit of failure. For illustration consider a blog application that may consist of the three components load balancer, application server together with a servlet, and a database. A component is composed of multiple software artefacts.

(Software) artefact A software artefact is any entity that is required for the execution of a component. This may be a binary, a shared library, an operating system package installed through the package manager, a software container, a `jar` file or anything the like.

Lifecycle handler The *lifecycle handlers* of a component are software programmes or scripts that define the basic life cycle actions of a component such as install, configure, and run the application component, but also health checking.

Application instance The deployment of an application results in an *application instance*. An application instance for application A is linked to at least one component instance for each component that belongs to A.

Component instance A *component instance* is an enacted component. Component instances are created through the lifecycle handlers associated with the respective component. Each component instance is run on a particular virtual machine VM_k and multiple component instances can be mapped to the same virtual machine.

Channel Components may be connected with each other using directed *channels*. Connecting two components with a channel imposes that at least one component instance from the source component will interact with at least one instance from the target component in the context of the same application instance. The concrete wiring between the source and target instances is subject to both the deployment and the scaling.

— 1 —

Introduction and Overview

The Executionware constitutes a fundamental part of the entire PaaSage system and its architecture. The primary purposes of the Executionware are *(i)* to enact interacting with the cloud providers through their respective and largely inhomogeneous APIs, in order to support the creation, configuration as well as tear down of virtual machines and virtual networks; *(ii)* to enact the deployment of application components such as load balancers, databases, and application servers across the created virtual machines; *(iii)* to support the monitoring of both virtual machines and application component instances by provisioning of appropriate sensors/probes and by supporting a reporting interface for applications; *(iv)* to support the aggregation of raw metrics coming from the probes to higher-level composite metrics, and the evaluation of any of these metrics according to conditions and thresholds; *(v)* to report back metric values to the Upperware and to store them in the Metadata-Database (MDDB).

This document presents an overview and documentation for the product release version of the PaaSage Executionware implementation. This chapter provides an overview on the Executionware and particularly presents its purposes and tasks within PaaSage (*cf.* Section 1.1), its overall mechanisms and architecture (*cf.* Section 1.2), while also introducing the software components. Afterwards, this chapter presents the changes that have been made compared to the initial prototype [6] (*cf.* Section 1.3). Finally, we present the structure of the remainder of this document in Section 1.4.

1.1 The Executionware in PaaSage

The Executionware constitutes a fundamental part of the entire PaaSage system and its architecture. The PaaSage architecture as described in deliverable D1.6.1 [2] defines the role of the Executionware in the PaaSage application lifecycle. Together with the user requirements as defined in deliverable D6.1.1 [7] these define the overall functionality provided by the Executionware.

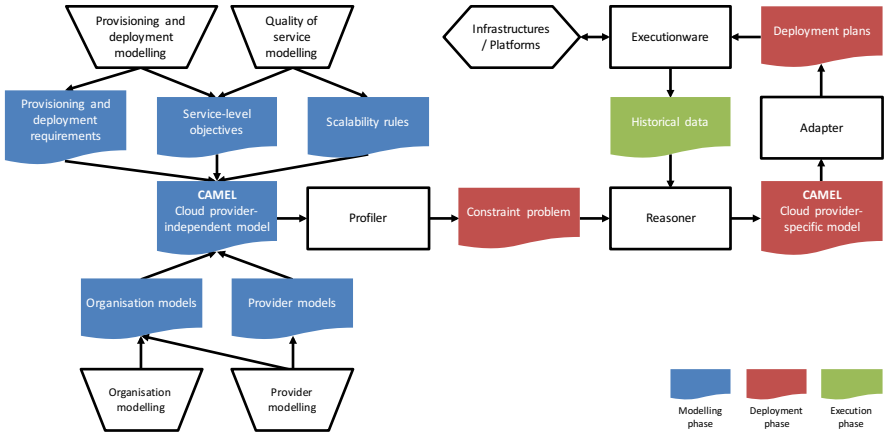

Figure 1.1: Main PaaSage components and life-cycle direction including data and model flow as deliverable D1.6.1 [2].

Summarising, the focus of the Executionware is three-fold: First, it is responsible for bringing applications to execution that have been modelled in CAMEL and whose deployment has been configured by PaaSage's Upperware component. Second, the Executionware is responsible for monitoring the execution of each individual component instance which leads to the monitoring of the overall application instance using defined aggregation methods. Third, the PaaSage architecture enables the Executionware to autonomously change the deployment of an application within certain boundaries and according to the application model. These changes comprise *e.g.*, adding new or removing existing component instances and virtual machines.

The initial PaaSage architecture has been specified in deliverable D1.6.1 together with a sketch of the Executionware architecture. The latter has then be refined in deliverable D5.1.1 [6] that also presented the initial prototype of the Executionware.

Figure 1.1 sketches the data flow and model flow between the individual PaaSage components at configuration, deployment, and execution phase. It points out that the application model is created in CAMEL [3]. Then, the deployment workflow ripples through the Upperware [4] and finally reaches the Executioware through the adapter. The adapter orchestrates the creation of virtual machines and the deployment of component instances over these virtual machines which is then executed by the Executionware.

Figure 1.1 also shows that selected data flows back to the Upperware. This is achieved using the monitoring and aggregation infrastructure of the Executionware. In addition, selected data is also stored in the metadata database. Again, it is the Upperware's adapter that defines the monitoring and aggregation to be used by the Executionware. In addition to that, the adapter configures the low-level

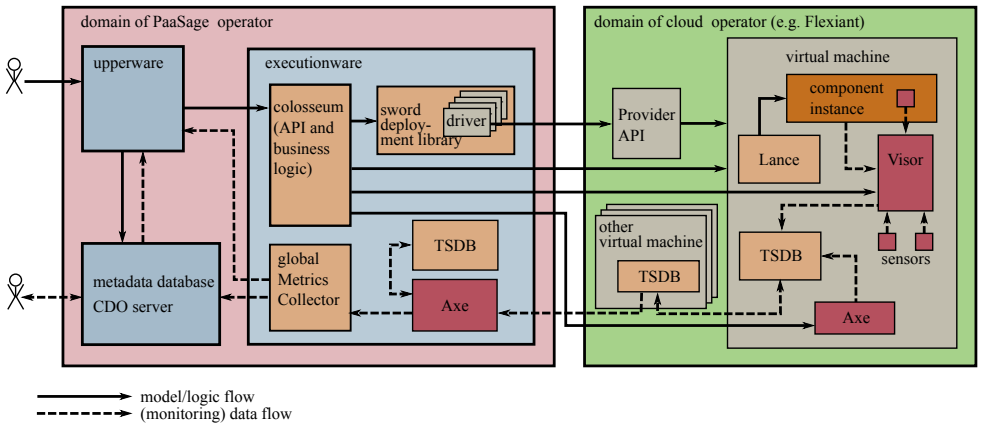

Figure 1.2: Architecture of the Executionware together with input to and output from other PaaSage components.

adaptation mechanisms (e.g. component scaling) and boundaries the Executionware is allowed to apply on an application instance.

1.2 The Executionware Architecture

Summarising Section 1.1 the Executionware needs to fulfil the following six tasks within PaaSage: *(i)* management (*e.g.*, creation) of virtual machines on different cloud providers; *(ii)* instantiation of components on those virtual machines and linking those component instances that belong to the same application instance on network level; *(iii)* collect monitoring data *(iv)* aggregate monitoring data, *(v)* evaluate monitoring data and apply adaptation rules, as well as *(vi)* relay monitoring data and information about scaling events to the Upperware.

Figure 1.2 shows the architecture derived for the Executionware. Much of the functionality *(i)*–*(vi)* is captured by components of the CLOUDIATOR tool set and some third party components. The `MetricsCollector` component is also able to realise the functionality (*(iii)*–*(iv)*), thus being an alternative to the respective CLOUDIATOR components, as well as complement the CLOUDIATOR tool with the capability to relay monitoring and event information to the Upperware module and the metadata database. In the following, we briefly introduce both the CLOUDIATOR tool in Section 1.2.1) as well as the `MetricsCollector` component in Section 1.2.2. Third-party tools are briefly described in Section 1.2.3.

1.2.1 CLOUDIATOR and its Tools

CLOUDIATOR[1] is a cross-cloud, multi-tenant tool that has been developed with the PaaSage project. In addition to cross-cloud deployment, it supports the `model@runtime` paradigm [16] and also comes with redeployment capabilities to support automatic as well as manual adaptations. Regarding auto-scaling support, CLOUDIATOR goes beyond basic threshold-based approaches and supports the full capabilities provided by the scalability rules language contained in PaaSage's CAMEL.

As presented in Figure 1.2, the functionality of CLOUDIATOR is split across a so-called home domain and possible multiple cloud domains. The components of the home domain form the static part of the infrastructure that is hosted by the PaaSage operator and is not necessarily executed in a cloud environment. The second group of CLOUDIATOR entities is brought out in the field together with application component instances. Their main purpose is the provisioning of a run-time environment for these component instances and to collect and store monitoring data. The following paragraphs briefly summarise the respective components and explain their features. Also, they reference the sections where the individual components are discussed in more detail. Apart from the components shown in Figure 1.2, the Executionware (in particular COLOSSEUM) comes with an extension that allows the user to trigger a full PaaSage workflow for a CAMEL model. This feature is sketched in Section 4.2.

Colosseum The COLOSSEUM component runs in the home domain and represents the central access point for any clients through a REST as well as a Web UI-based interface (*cf.* Section 2.1.2). Clients may either be human operators or third party tools including the components of the Upperware. COLOSSEUM contains a set of registries that store information about cloud providers, created virtual machines, components, and component instances. Colosseum and its API is presented in detail in Section 2.1.

Sword SWORD is a library that provides an abstraction layer for the various cloud providers. In particular, it encapsulates the differences between them with respect to terminology and technology such as the provisioning of floating IPs, passwords, and access to virtual machines. Sword is introduced in more detail in Section 2.2.

Lance LANCE runs in the cloud domain. In particular, Colosseum will deploy one instance of Lance on each virtual machine it creates. Lance is responsible

[1] `https://github.com/cloudiator/`

for executing the life-cycle of the component instances to be installed on the virtual machine. Hence, Lance executes the scripts to download, install, configure, and start instances, as well as those for stoping the component and those for updating the configuration when the set-up of the application instance changes, for instance, because a new downstream component has been added. Lance is depicted in detail in Section 2.3.

Visor VISOR runs in the cloud domain and is responsible for collecting monitoring data from virtual machines and component instances. In particular, Colosseum will install an instance of Visor in any virtual machine it creates. Whenever Colosseum is requested to monitor certain aspects of an application and/or virtual machine, it will connect to the Visor of that virtual machine and request the installation of a sensor at Visor together with a time interval. Visor will then invoke the sensor at the requested interval and send the data to the configured time-series database (TSDB) (*cf.* Section 1.2.3). In addition to that, component instances can connect to Visor and report application-specific metrics. Visor, its default set of sensors, as well as its API are discussed in Section 3.1.

Axe AXE is a two-purpose component that runs partially in the home and partially in the cloud domain. Its first task is to post-process the data collected by the visor component. In particular, Axe is capable of executing aggregation functions on the monitored data such as computing amongst other averages, medians, and quantiles. It reports selected metrics to the Upperware via the `MetricsCollector` (*cf.* Section 1.2.2). In addition to that, Axe is concerned with evaluating conditions on the measured and aggregated data. It will then report violations of these conditions to the Upperware via the `MetricsCollector` or even trigger changes in the application instance, *e.g.*, by adding new virtual machines and component instances (scale out). Section 3.3 presents AXE in detail.

1.2.2 The `MetricsCollector` Component

The `MetricsCollector` component is a multi-purpose component. The respective software runs as a daemon that can be operated either in local or global mode. In local mode it replaces parts of the functionality of CLOUDIATOR's AXE; hence, `MetricsCollector` instances running in local mode are optional and dependent on the configuration of the Executionware. In contrast, in global mode, the `MetricsCollector` component provides a link to the Upperware components and enables registering metric data as well as violations of metric conditions, i.e., events in the meta-data database (CDO server [3]). The

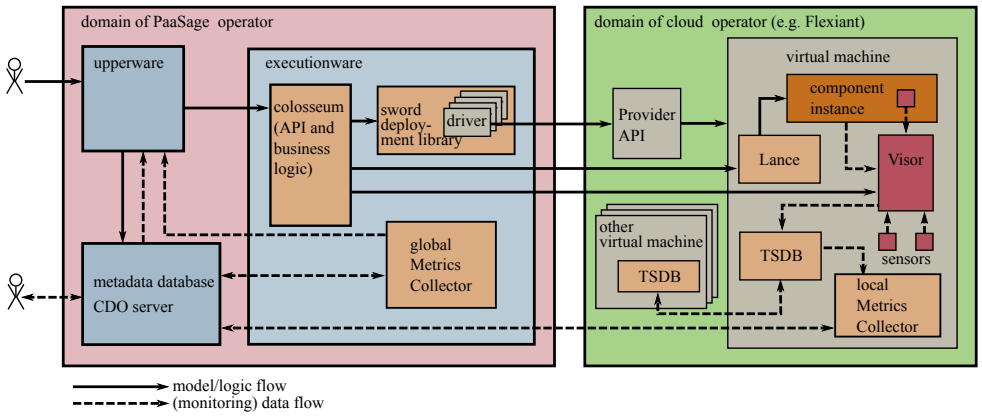

Figure 1.3: Architecture of the Executionware with `MetricsCollector` operating in global and local mode.

`MetricsCollector` is subject to Section 3.4. Figure 1.3 sketches the alternative architecture with `MetricsCollector` running in both local and global mode.

Local mode When `MetricsCollector` local mode is enabled in the Executionware, Colosseum will install and start one `MetricsCollector` instance at each virtual machine it creates. Just as AXE, it will read data from the TSDB (*cf.* Section 1.2.3) and perform aggregations on the measured data. In contrast to AXE, however, it will directly write the results to the meta-data database. As such, each `MetricsCollector` instance will connect to the CDO server of the system to report the aggregated metric values.

Global mode The usage of a `MetricsCollector` in global mode is twofold. When AXE is used, AXE uses the `MetricsCollector` instance to report the configured metrics (as well as respective events) to the Upperware and the meta-data database. If local `MetricsCollectors` are used instead, the global `MetricsCollector` will listen to new measurements added to the CDO server by the local `MetricsCollector` instances. It will then apply the necessary aggregation functions and write the results back to the CDO server, but also relay them to the Upperware through a pub/sub mechanism based on ZeroMQ[2].

[2] `http://zeromq.org/`

1.2.3 Third Party Components

Beside the components described above that have been developed in the scope of the project, the Executionware relies on two components that are provided by third parties: the TSDB as well as the registry service.

TSDB The time-series database is used by VISOR to buffer measured data and by AXE and `MetricsCollector` (in local mode) respectively to retrieve data necessary for their aggregations. AXE also uses the TSDB for buffering intermediate results. Currently, the Executionware supports KairosDB [3] as TSDB, but other implementations have been considered. The selection process is detailed in Section 3.2.

Registry service For being able to track the status of the installed and started component instances and their wiring, LANCE relies on a registry service where it puts status information and network configurations. Currently, LANCE supports two implementations for this service: a Colosseum-internal one with volatile state and a persistent one based on the external directory server `etcd`[4]. The registry's usage is detailed with the description of Colosseum (*cf.* Section 2.1) and LANCE (*cf.* Section 2.3) respectively.

1.3 List of Changes from Initial Prototype

Compared to the initial prototype presented in M18 in D5.1.1 [6], several things have changed. The most noticeable is the fact that the Executionware has abandoned the use of Cloudify v2.7 and replaced the deployment engine with a custom implementation, CLOUDIATOR, that has been released as an open source software library[5]. The reason for that move was two-fold: First, with the release of the M18 prototype, GigaSpaces—the maintainer of Cloudify—released version 3 of Cloudify and discontinued support for version 2.7 that the prototype had been using. Second, the new release contained concepts that were incompatible with the PaaSage architecture. Also multiple required features such as the user interface were only available in the commercial version. We decided to implement CLOUDIATOR from scratch in order to overcome limitations that existing tools have and to avoid their vendor lock-in as well as the instabilities of many open source tools such as Apache Stratos[6] and Apache Brooklyn[7].

[3] `https://github.com/kairosdb/kairosdb`
[4] `https://github.com/coreos/etcd`
[5] `https://github.com/cloudiator`
[6] `http://stratos.apache.org/`
[7] `https://brooklyn.incubator.apache.org/`

In addition to that, the feature completion has advanced beyond what was presented in M18. While almost all desired features have been integrated and implemented, some of them such as the Windows support are in a very early stage and will have to be finalised and hardened in the upcoming Year 4 of the project.

1.4 Structure of This Document

The structure of this document has been widely introduced while describing the individual components. Hence, the following Chapter 2 introduces Colosseum, SWORD, LANCE, whereas Chapter 3 presents monitoring-related aspects such as VISOR, the TSDB selection, AXE, as well as the `MetricsCollector` in detail. The remaining parts of the document address further aspects of the Executionware in Chapter 4 such as the operation of testbeds and the provisioning of an access point for users such as the social network [5]. Finally, Chapter 5 concludes the document with an overview of the components, and their download location.

— 2 —

CLOUDIATOR

Overall, CLOUDIATOR is a multi-tenant-capable Web-based software service. Its features can be separated into registries, deployment functionality, automatic adaptation, and the specification of monitoring requirements. This chapter gives a more extended description on three of the CLOUDIATOR tools that have already been introduced in Section 1.2.1: COLOSSEUM as the entry point to the system; SWORD providing the resource allocation functionality; and LANCE for the life-cycle handling of component instances. The overall CLOUDIATOR component layout is also summarised in Figure 2.1.

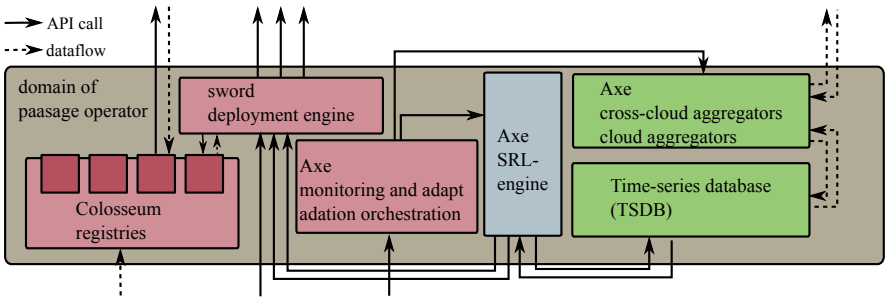

Figure 2.1: Detailed CLOUDIATOR components running in the domain of the PaaSage operator.

2.1 COLOSSEUM

COLOSSEUM is the central access point to interact with CLOUDIATOR and hence, the Executionware. In consequence, it offers APIs that clients as well as higher-level components such as the Upperware can use. Moreover, COLOSSEUM contains several registries that contain intermediate data necessary to control and surveil actual deployments. The following sections first introduce the various

registries (*cf.* Section 2.1.1) and then present the REST-based as well as the browser-based API in Section 2.1.2.

2.1.1 Registries

The data stored in the registries lay the ground for the management, access to, and comparison of cloud providers. They are essential for the deployment (*cf.* Section 2.2) and monitoring features (*cf.* Chapter 3). So far, CLOUDIATOR and with it COLOSSEUM contains four different registries: *(i)* The *cloud registry* stores offerings of cloud providers. This includes the type of cloud platform offered, the data centres and availability zones offered by that provider, the virtual machine types (flavours) and operating system images available at each of these system levels. Additional geographical location information can be attached to each data centre. *(ii)* The *specification registry* stores abstract properties of cloud providers. This includes generic virtual machine specifications consisting of number of cores and amount of RAM. This registry, also supports an operating systems hierarchy that for instance states that `Ubuntu 14.04` belongs to the `Ubuntu` family which in turn belongs to the class of `Linux` operating systems. The entries of the specification registry are linked to these of the cloud registries where applicable. *(iii)* The *credential registry* holds cloud access credentials for each user of CLOUDIATOR needed for SWORD to access the cloud providers. The kind of credentials stored vastly depend on the underlying cloud platform. *(iv)* The *component registry* contains components and their description (*e.g.* lifecycle management information) which can be assembled to applications. These applications can then be instantiated (cf. Section 2.3). Each of the registries is multi-tenant capable, meaning that any entry is bound to a CLOUDIATOR tenant.

2.1.2 Usage and APIs

CLOUDIATOR can be used in two ways by the end users. First, abstract requirements regarding the virtual machines can be specified in a cloud-independent way relying on CLOUDIATOR's simple reasoning functionality.

The second approach is to specify in a fine-grained way which concrete virtual machine flavours shall be used on what cloud and with what image. In this case, it is necessary to combine CLOUDIATOR with a more powerful reasoning mechanism and a modelling approach. Obviously, this is the approach the platform is used within a PaaSage setting. Here, the modelling is done through a CAMEL model followed by a multi-step reasoning process that eventually yields a deployment plan including monitoring, aggregation and scalability rules.

The following sections briefly introduce the REST-ful API offered by COLOSSEUM as well as the Web-based user interface.

```
GET /api/hardwareOffer/2
{
  "numberOfCores":4,
  "mbOfRam":4096,
  "localDiskSpace":20000
}
```

(a) Sample REST response of a call to retrieve details about a cloud-operator independent hardware description

```
GET /api/hardware/2
{
   "cloud":1,
   "hardwareOffer":2,
   "remoteId":"939c4993-8562-42af-a80c-d8829863d433",
   "locations": [1, 2 ],
   "cloudCredentials": [1 ]
}
```

(b) Sample REST response of a call to retrieve details about a cloud-operator specific hardware description referencing the operator-independent description

```
POST /api/virtualMachine
{
   "name":"scalarm_vm_1",
   "cloud":1,
   "image":4,
   "hardware":2,
   "location":1
}
```

(c) Sample REST call to start a virtual machine with a dedicated image, at a dedicated cloud provider, and with a specific operating system image.

Figure 2.2: Three sample calls to the COLOSSEUM API in order to retrieve hardware offers, hardware types, and eventually create a new virtual machine.

REST API

COLOSSEUM provides a RESTful API that ranges from storing cloud providers, their offers and locations over the handling of hardware specifications to the deployment and monitoring of applications, components and their respective instances. This is the API that is used by the adapter of the Upperware. Figure 2.2 shows three example calls to the REST API. These are only supposed to give a

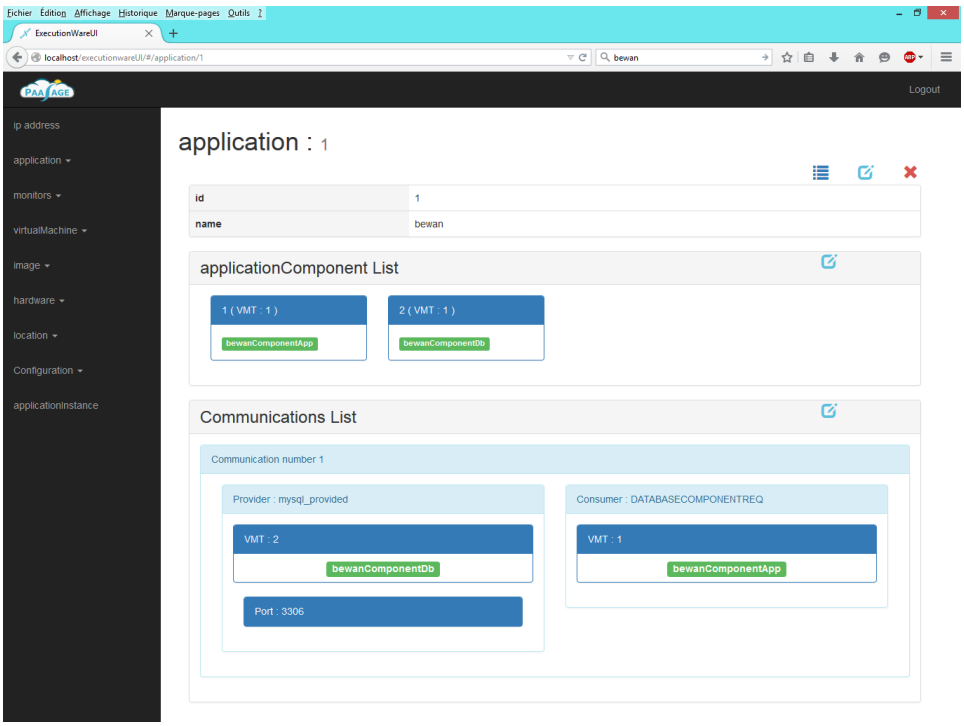

Figure 2.3: Screenshot of the ExecutionwareUI showing an application configuration

brief overview on the API and how it is supposed to be used. The full API is documented in COLOSSEUM's GitHub page[1]

In addition to specifying monitoring information that is collected and evaluated for the adaptation functionality, a user can specify further monitoring requirements. Here, he defines sensors that collect the necessary data and instructions that define how these raw metrics shall be aggregated to higher-level metrics. This data is provided to the clients of CLOUDIATOR through the COLOSSEUM API. The monitoring interface is further discussed in Chapter 3.

User Interface

The ExecutionwareUI provides a RESTFul, browser-based client for the COLOSSEUM API (*cf.* Figure 2.3). The main goal is to provide users with human-readable information from this API such that it is possible to help developers and testers of the platform to tightly follow the steps the Executionware performs during debugging and execution.

[1] `https://github.com/cloudiator/colosseum/blob/master/documentation/api/README.md`

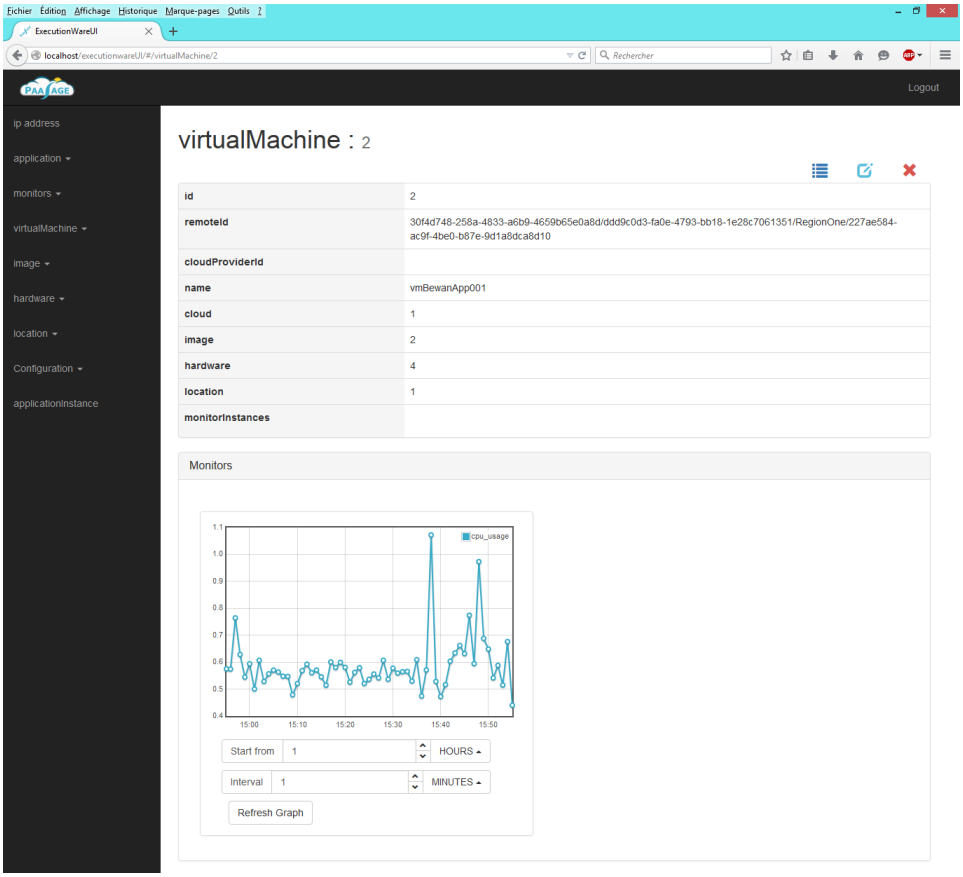

Figure 2.4: Screenshot of the ExecutionwareUI showing monitoring data for a deployed virtual machine

While the ExecutionwareUI is in general capable of modifying the entries of the system, this use case is currently not exploited within PaaSage, as manual intervention could lead to inconsistencies between the Executionware's and the Upperware's state. This functionality is therefore mainly used for testing purposes.

A particular feature of the ExecutionwareUI is that it provides access to monitoring that is shown as a graph (*cf.* Figure 2.4).

Technical background The ExecutionwareUI runs in a browser and is widely based on angular.js[2]. The web site rendered by angular.js displays information provided through AJAX request. In order to display graphs, the client implementation uses the Flot library[3].

[2] `https://angularjs.org/`
[3] `http://www.flotcharts.org/`

```
"api" :{
   "api" :"colloseumapi",
   "crud" :["list", "get", "put"],
   "form" :{
      "name" :"string",
      "internalProviderName" :"string"
   },
   "list" :["id", "name" ],
   "details" :["id", "name", "internalProviderName" ],
   "tostring" :"name"
}
```

Figure 2.5: Sample configuration of the ExecutionwareUI that enables listing, retrieval, and putting entities to the `colosseumapi`, but would not allow the creation or deletion of entities.

In order to cope with the distributed implementation and the fast changing interfaces of an ongoing research project such as PaaSage, the ExecutionwareUI does not directly rely on the COLOSSEUM API, but uses a configurable intermediate proxy that receives requests from the browser implementation and relays respective messages to COLOSSEUM. This architecture supports to host all static content outside COLOSSEUM and is also capable of realise caching.

Configuration The flexibility for coping with changing conditions and interfaces is reflected in the configuration capabilities of the PHP-based forwarding proxy component. This feature targets the ability to change the structure of some `apiObject` (representing entities like images or virtual machines) and to (temporary) enable (disable) particular actions such as editing and deleting entities of `apiObject`. Figure 2.5 presents a sample configuration statement. Here, the entities shall be shown and can be modified; yet, they cannot be created or deleted. Furthermore, the `list` view filters out `internalProvider` field.

2.1.3 Further Features

For CLOUDIATOR can also be used outside PaaSage, as COLOSSEUM supports further features. In particular, it contains a built-in discovery mechanism that enables providing the data for the registries (cf. Section 2.1.1) in an automated way: Whenever a user registers credentials for a cloud provider the COLOSSEUM discovery mechanism will fill the cloud registry for that cloud provider. In addition, it will connect this information to the specification registry with the abstract cloud properties.

As it may not be possible to retrieve all information via the providers' APIs, the COLOSSEUM API allows the manual completion of entries. In the future,

we plan to rely on using meta-information providers, such as CloudHarmony[4]. These provide additional information such as the actual geographical location of the cloud provider-specific location (either region or availability zone).

2.2 SWORD

SWORD comprises an abstraction layer over the APIs of the various cloud providers. Its implementation is widely based on the OpenSource Apache jclouds library. Yet, in addition, it comes with a custom implementation for the Flexiant FCO platform including the latest FCO v5 release.

Based on the information received at the virtual machine creation, SWORD selects the right cloud provider and contacts his API through jclouds to issue a start command. The necessary cloud-specific image and hardware flavour identifiers are retrieved from COLOSSEUM's registries. After a successful boot, SWORD will assign a public IP address to the virtual machine.

Once this has been done, SWORD logs in to the virtual machine using SSH (for Linux machines) and WinRM (for Windows machines) using the XebiaLabs Overthere Library[5]. Afterwards, it starts installing system components. At the time being, these comprise VISOR, an instance of the TSDB, and LANCE. Furthermore, an aggregator of AXE or a `MetricsCollector` are installed on the virtual machine. The installation of the application itself, and hence the execution of LANCE actions, is triggered by deployment requests to COLOSSEUM and is not the task of SWORD.

2.3 LANCE

LANCE is CLOUDIATOR's life-cycle agent who is responsible for executing the life-cycle actions of all component instances running on that virtual machine. This task comprises the execution of installation and configuration scripts provided by the users through CAMEL and hence COLOSSEUM, but also the start and surveillance of the component instances; again through user-specified scripts. When the application set-up changes, *e.g.*, because a new application server had been added to the system, then other components linked to that component instance (*e.g.*, the load balancer) are notified and user-provided update scripts will be executed.

It is important to note that LANCE does not execute any magic, but solely performs the tasks provided in the application model. Hence, a missing communication link in the application model may lead to failures during deployment

[4] `https://cloudharmony.com/`
[5] `https://github.com/xebialabs/overthere`

as the necessary network ports are not open. Also, determining the number of component instances and on which virtual machine to place them is not the responsibility of LANCE or any other CLOUDIATOR component. Instead, those decisions have to be taken outside CLOUDIATOR. In PaaSage, this is the task of the Upperware.

Hence, LANCE has the primary task to read a component specification, reserve a separate space for it on the virtual machine, and to start the component by running the lifecycle handlers. Currently, two different mechanisms are available in order to isolate component instances running on the same virtual machine: the first approach solely relies on the file system and provides each instance an own directory on the file system. This is the approach used for virtual machines running Windows. For Linux-based virtual machines, by default Docker[6] containers are used in order to isolate component instances from each other.

[6] `https://www.docker.com/`

— 3 —

Monitoring and Auto-scaling

For supporting in-depth analysis of existing deployments, several requirements have to be considered: *(a)* The fact that on the one hand, the monitoring of large-scale applications does generate huge amounts of data and on the other hand cloud providers usually charge for network traffic that leaves their data centre gives motivation that as much of data processing shall happen within the domain of individual cloud providers. *(b)* In order to avoid single points of failures, the architecture of a monitoring solution should not rely on a centralised approach, but rather favour distributed approaches with no central entity. *(c)* For the amount of monitoring data usually increases with the number of allocated *virtual machines (VMs)*, the resources assigned to monitoring shall increase with the size of the application. *(d)* The operators of a cloud application may discover that they have to monitor further high-level or even low-level metrics or need monitoring to happen at a higher resolution. Hence, it is necessary that monitoring properties can be changed also after an application has been deployed. *(e)* The same considerations that hold for monitoring, have to hold for scaling rules. In addition, it is necessary that rules can be defined in a generic way without having to know the exact number of instances per component in advance. *(f)* The monitoring platform has to be able to capture application-specific metrics.

With respect to PaaSage, these considerations hold as well, but in addition to that, the monitoring data has to be relayed to the Upperware such that it can take decisions and adapt the deployment as needed. In this chapter, we introduce the entities of the Executionware that are related to monitoring. These include the CLOUDIATOR components VISOR (*cf.* Section 3.1) and AXE (*cf.* Section 3.3), but also the `MetricsCollector` component (*cf.* Section 3.4). Furthermore, Section 3.2 discusses various TSDBs and presents the discussion we applied for the Executionware at the time being.

3.1 VISOR

In order to be able to gather the raw monitoring data from the VMs and component instances, we introduce VISOR as a monitoring agent to the remote cloud domain. Just as LANCE, VISOR is deployed on every VM and provides a remote interface COLOSSEUM uses in order to configure a particular VISOR instance. This allows VISOR to be adopted to the application and to only collect the metrics actually required, thus saving space and bandwidth. VISOR supports the capturing of monitoring data on a per component instance basis as well as on a per-VM basis. The former is done by exploiting the fact that, at least for Linux applications, all component instances are run inside a Docker container and the resource consumption can be retrieved on a per-container basis. The latter is achieved by sensors monitoring basic system properties on virtual machine level, e.g. by accessing system properties such as CPU load.

VISOR produces raw monitoring data through a set of installed sensors. A user or the Upperware can install sensors by defining monitoring requirements or scalability rules through the COLOSSEUM interface (cf. Section 2.1). COLOSSEUM will then forward the installation request to the VISOR running on the specific virtual machine or to all virtual machines in case multiple of them are affected from a single interface access. VISOR contains a set of well-known sensors for measuring system parameters such as CPU and RAM utilisation as well as I/O rate. The sensors available by default are subject to Section 3.1.1. Each sensor can be configured to have a dedicated interval for which the data shall be collected as well as a measurement context that defines whether the monitoring shall capture the entire virtual machine or only a particular component instance on that virtual machine. In order to support custom metrics, VISOR supports the implementation of custom sensors, by providing an easy-to-implement Java interface. It exploits the dynamic class loading properties of Java in order to be able to add those implementations at runtime.

For supporting application-specific metrics that can only be retrieved from within an application such as the length of queues or the degree to which buffers have been filled, VISOR offers a text-based interface over a network socket with a well defined port number. Applications can push their metrics data to an interface which is compatible with the carbon daemon of graphite[1], thus allowing an easy migration to VISOR. An example thereof is subject to Section 3.1.2.

Finally, VISOR forwards all measured monitoring data to the aggregation and rule processing sub-system. This version of the Executionware uses a TSDB to store the data. The core part of VISOR, however, is not dependent on the data

[1] `http://graphite.readthedocs.org/en/latest/carbon-daemons.html`

Table 3.1: Table of default sensors implemented for VISOR and their context parameters

Sensor Class	Context Parameter	Description	Example Value
`CpuUsageSensor`	`N/A`		
`MemoryUsageSensor`	`N/A`		
`DiskIoReadSensor`	`fs_device` `unit`	Mounting point of a disk Unit of measurement, e.g. MBytes/s	sda mb
`DiskIoWriteSensor`	`fs_device` `unit`	Mounting point of a disk Unit of measurement, e.g. MBytes/s	sda mb
`DownloadBandwidthSensor`	`net_device` `unit`	Network interface Unit of measurement, e.g. KBytes/s	eth0 kb
`FreeDiskSpaceSensor`	`fs_root` `unit`	File system root path Unit of measurement, e.g. GBytes	/ gb

`NetworkLatencySensor`	`host`
	`port`
	`loops`
		Hostname
		Port number
		Number of measurements
			www.google.com
			80
			3

`NfsAccessSensor`	`nfs_root`	NFS mounting point	/media/nfs
`RxBytesSensor`	`net_device` `unit`	Network interface Unit of measurement, e.g. MBytes	wlan0 mb
`RxPacketsSensor`	`net_device`	Network interface	eth0
`TxBytesSensor`	`net_device` `unit`	Network interface Unit of measurement, e.g. MBytes	wlan0 mb
`TxPacketsSensor`	`net_device`	Network interface	eth0
`UploadBandwidthSensor`	`net_device` `unit`	Network interface Unit of measurement, e.g. KBytes/s	eth0 kb

sink so that other approaches such as the use of data streams are thinkable as well.

3.1.1 Available Sensors and Probes

VISOR comes with a set of pre-defined sensors that can be grouped into system sensors, network sensors, and others. Most sensors rely on the System Information Gatherer and Reporter (Sigar) library and API[2]. SIGAR provides unified access to operating system metrics [15], but at the same time abstracts from the concrete operating system. The following three paragraphs overview system sensors, network sensors, and file system sensors. Table 3.1 summarizes all

[2] `https://github.com/hyperic/sigar`

implemented sensors and shows their configuration by documenting the context parameters and depicting example values.

System Sensors

Memory (RAM) Usage The memory usage sensor measures the current memory used by the system in percentage. For this task it uses the Java class OperatingSystemMXBean[3], allowing the measurement on all operating systems supporting Java.

CPU Usage The CPU usage sensor measures the load of the CPU in percentage, by calculating the average load across all cores installed in the system. Just as the memory usage sensor it therefore relies on the OperatingSystemMXBean, allowing its measurement on various operating systems.

Network Sensors

Average Network Latency Network latency can be measured two-fold: Latency can be assessed either by the time a data packet requires to reach its destination (*i.e.*, one-way), or by additionally taking into account the time it takes back from destination to source (*i.e.*, round-trip). Intuitively, the ping utility is one of the immediate choices to measure a round-trip based latency [4] through the Internet Control Message Protocol (ICMP). Yet, the round-trip time measured by ICMP is not representative, as it is barely affected by host load. As such, ping-based latency is not representative for real-world application latency. For this reason, the network sensor bases on the common Transport Control Protocol (TCP) and relies on a simple socket connection established via native Java methods to obtain the latency measurement. Users can set the following parameters in the configuration file: destination address, port number, and number of measurements (cf. Table 1). The sensor then computes an average over all measurements and reports this as a single measurement.

TxBytes and RxBytes This sensor implements logic to report on transmitted and received bytes of a given network interface. A proxy to the class `NetInterfaceStat` of the Sigar API was implemented to monitor data. The respective class provides accumulated data for transmitted (*TxBytes*) and received (*RxBytes*) bytes, respectively. For example, the command-line utility `ifconfig` is used on Linux platforms to monitor transferred bytes. Separate sensors for transmitted and received data are provided.

[3] `https://docs.oracle.com/javase/8/docs/jre/api/management/extension/com/sun/management/OperatingSystemMXBean.html`

[4] `http://manpages.ubuntu.com/manpages/trusty/man8/ping.8.html`

Bandwidth This sensor implements functionality to measure upload and download data rates of a given network interface. We reuse the implementations for the *TxBytes* and *RxBytes* sensors to compute the bandwidth. Those values are then used to compute the bandwidth per second. For each connection type, i.e. upload or download, an individual sensor is implemented.

TxPackets and RxPackets This sensor reports on transmitted and received network packages of a given network interface. This information is again retrievable through the `NetInterfaceStat` class of the Sigar API. The sensor follows the implementation of the *TxBytes* and *RxBytes* sensors, but instead of monitoring transferred bytes, it is monitoring transferred packets.

File System Sensors

Free Disk Space This sensor reports on the available disk space for a given file system. To achieve a unified monitoring solution, we rely again on the Sigar API. The API implements the class `FileSystemUsage`, that offers a proxy to native utilities such as `df` on Linux systems. The file system's root path can be configured via the user-defined context at run-time.

Disk I/O This sensor reports on the I/O performance of a given disk. The class `DiskUsage` of the Sigar API is employed to assess relevant values for disk reads and writes per second. For example, the command-line utility `iostat` is used on Linux systems to obtain this data. The functionality is split into two individual sensors that measure disk reads and writes, respectively. Users are able to select the disk and unit of measurement (e.g., MBytes/s).

NFS Connection Status This sensor checks if a network file system (NFS) is mounted and accessible. A client connection is established to network shares using NFS clients provided by the Sigar API. Our implementation handles both NFSv2 and NFSv3. Users have to set NFS mount point for checking via context parameters.

3.1.2 Application-specific Probes

In order to demonstrate the capabilities of the Executionware and VISOR in particular to handle custom metrics, we briefly describe the custom monitoring infrastructure as shipped together with the LSY application [8]. While this monitoring infrastructure duplicates some aspects of the Executionware monitoring system, as it contains legacy components, it is also a striking demonstration of the flexibility of the PaaSage monitoring infrastructure.

When implementing the LSY monitoring system, the goal was to implement a configurable and extensible solution, which will run in a java virtual machine (JVM). The requirements for such an approach include *(i)* an abstraction over metrics sources—the implementation should allow the user to collect metrics from any kind of source with any kind of data format. *(ii)* an abstraction over metrics target—the implementation should allow the user to pass the metric to any kind of database or external system. *(iii)* The implementation should allow the user to process probes—push them through some logic *e.g.* filtering, aggregating etc. *(iv)* The implementation should allow the user to apply a different processing algorithm over the same probe. *(v)* The implementation should allow the user to mock metrics, *i.e.*, to change probe values at runtime

Concept and Components

The main idea of the LSY monitoring system is to to describe the processing of the metric data flow with streams with a strong focus on the Reactive Manifesto[5]. This lead to the selection of Akka Streams[6] which provides the end user a solid API to implement the logic and internally relies on the Reactive Streams interfaces.

At run-time, data is collected from all instances of all LSY system components. All of them run with a JVM and metrics can be retrieved using JMX. These data tuples are tagged and then relayed to further downstream processing components where the data is aggregated.

The computations and aggregations on the data flow are realised by splitting the data flow into sub flow(s) and then applying dedicated business logic for each of these flows.

All data that is eventually collected and processed is afterwards stored in InfluxDB (*cf.* Section 3.2). From there it is fetched and then shown using the Grafana Dashboard[7].

Integration in Executionware

Obviously, an integration of this monitoring mechanism into the Executionware can happen on multiple levels. In an initial approach, we deploy the entire monitoring framework as part of the LSY application. Then, instead of writing the data into InfluxDB it is relayed to the Executionware's TSDB using VISOR.

[5] `http://www.reactivemanifesto.org`

[6] `http://doc.akka.io/docs/akka-stream-and-http-experimental/1.0/scala/stream-introduction.html`

[7] `http://grafana.org`

3.1.3 Application specific probes for Scalarm and HyperFlow engines

Execution engines for eScience use case applications provide application specific probes and metrics that are used by classes of applications supported by SCALARM and HyperFlow, i.e. data farming applications and large-scale scientific workflows.

Scalarm Platform Sensors

System Simulations Throughput This sensor reports on the current throughput - measured in completed simulations per second - of the currently running data farming experiment. First, Scalarm calculates the throughput of individual Workers (worker throughput) by taking into account the number of finished and still running simulations on the particular worker, this value is divided by the time of Worker activity. The total System Simulations Throughput is the sum of all worker throughputs.

Response Time Of Experiment Manager This sensor reports on the average response time - measured in milliseconds - of the Experiment Manager service. The calculation of the average is performed by Scalarm and uses data measured and stored internally by the Scalarm platform.

Information Service Response Time This sensor reports on the average response time - measured in milliseconds - of the Information Service service. The calculation of the average is performed by Scalarm and uses data measured and stored internally by the Scalarm platform.

HyperFlow sensors

stage Workflow stage, as reported by the workflow engine. This metric informs about the stage of the workflow that is currently processed. It can be used to trigger autoscaling actions when a certain stage of workflow is reached and a required number of VMs is known from the scheduling plan provided by HyperFlow.

tasks, tasksProcessed and tasksLeft The number of tasks in the workflow: total, processed and left, respectively. These metrics can be used together to compute the composite metrics related to the progress of workflow execution.

outputsLeft The number of remaining outputs (results) in the workflow. Once all outputs are completed, the workflow is considered as finished.

nConsumers Number of active workers that are connected to the AMQP (Advanced Message Queuing Protocol) queue. This allows to measure the actual workers (executors) that have started and stressfully registered with the RabbitMQ[8] queue. This number may be different from the number of worker VMs, since for some reasons not all worker processes on VMs may have started, or there could be a failure during execution.

3.1.4 Client libraries for Visor

To facilitate development of application specific probes, we have developed generic clients for Visor in .NET, Java and JavaScript. These small libraries can be used to report metrics to Visor at configurable intervals.

All these libraries are available on GitHub repository[9] and have been used by the use cases of LSY and AGH.

3.2 TSDB Selection

A key element when computing higher-level metrics especially over larger time-windows is the need to buffer raw monitoring data. TSDBs have been designed to store timestamped data in an efficient way and also to provide quick access to the stored data. Many TSDB implementations support applying functions on stored data right out of the box, making them a perfect match not only for buffering, but also for aggregation (*cf.* Section 3.3). The following subsections first introduce the requirements towards a TSDB in Section 3.2.1. The requirements are based on the outlined monitoring and buffering strategy in COLOSSEUM. Afterwards an overview on three popular TSDBs, KairosDB, OpenTSDB and InfluxDB is presented. In Section 3.2.5 we argue why the current version of CLOUDIATOR makes use of KairosDB, whereas Section 3.2.6 presents the installation process.

3.2.1 Requirements

With respect to our requirements the buffering and therefore the TSDB approach needs to be able to work with limited resources, have no single point of failure and increase available resources when more VMs are being used. In order to cope with these requirements, we use the following approach: from each VM acquired for an application, we reserve a configurable amount of memory and

[8] `https://www.rabbitmq.com/`

[9] `https://github.com/dbaur/monitoring-agent-telnet-client-java`, `https://github.com/dice-cyfronet/monitoring-agent-telnet-client-js`

storage (e.g. 10%) that we further split between a *local storage area* and a *shared storage area*. Both storage areas are managed by a TSDB instance running on the VM. The VISOR instance running on this VM will then feed all monitoring data to the TSDB. The TSDB will store data from its local VISOR instance in the local storage area and further relay the data to other TSDBs shared storage area. This feature avoids that a TSDB becomes a single point of failure, but still enables quick access to local data. In order to keep network traffic between cloud providers low, any TSDB will only select other TSDBs running in the same cloud to replicate its data. If not enough instances are available to reach the desired replication degree, the maximum possible degree is used. Hence, this concludes to a ring-like topology that has been introduced in peer-to-peer systems [1] and is also used by distributed databases [14].

3.2.2 KairosDB

KairosDB[10] is a widely used TSDB that has continuous development. It is efficient in the storage of huge amounts of timeseries (TS) data as well as their respective querying via aggregation functions. It currently supports the following aggregation functions: average, standard deviation, divisor (divides each TS value with a specific constant), histogram (actually calculates a percentile for a particular set of TS data), least_squares (returns two TS data/points which best fit the line that characterizes a set of TS points), max and min, rate (calculates the rate of change between two TS points), sum (calculates the sum of values for a TS set).

While histogram is stated to be supported, it was impossible to actually use it due to bugs in the client software.

The user needs to indicate the time interval in order to enable KairosDB to generate the set of TS points that should be used for the aggregation. KairosDB further supports grouping based on tags, time ranges or values as well as the filtering based on tags. Other advantages of KairosDB are: (a) it provides a REST interface for the publishing and querying of TS data; (b) it supports distributed configurations with configurable replication factors through the use of the Cassandra store.

A basic graphical UI is also offered which enables the posing of queries and the presentation of the results in a graph, while the query performed in a form-based manner is transformed and represented in a JSON format which can then be exploited for calling the respective method of the REST API offered. It is also possible to replace the default dashboard by external dashboards like Grafana.

[10] `https://github.com/kairosdb/kairosdb`

KairosDB architecture offers the possibility to use different datastores. In particular KairosDB supports three underlying stores: Cassandra, HBase and H2, where the latter is the default.

The main disadvantage of KairosDB is that it does not support the automated background generation of aggregations. This means that it is up to the user to either create the code which will calculate the aggregated value for the particular composite metric at hand and store it inside the KairosDB or that the user can exploit an aggregated query to just calculate this aggregated value and report it. The second way is of course only convenient when there is one aggregation level. However, when more aggregation levels are involved, then only the first way (or a mixture of first way for all aggregation levels apart from the top one to be handled by the second way) is suitable as there is a need to at least store the measurements produced for the intermediate aggregation levels.

3.2.3 OpenTSDB

The TSDB OpenTSDB[11] was one of the first exponents of TSDBs. It still has an ongoing development with the current version 2.1.0. The aforementioned TSDB KairosDB is a fork of OpenTSDB version 1.0.0; yet both TSDBs focus on different goals.

OpenTSDB provides a basic set of querying and aggregation features however in a less sophisticated way as KairosDB does. OpenTSDB TS data can be accessed via telnet, REST API or a command-line client.

According to the presentation of TS data OpenTSDB differs considerably to KairosDB. KairosDB aims to separate data and presentation while OpenTSDB focuses on the presentation of server-side generated graphs. This approach improves the representation of graphs by using, *e.g.* interpolation but it limits the flexibility for processing the TS data. To display the TS data a built-in dashboard is provided which can also be replaced by various third party dashboards.

OpenTSDB uses the distributed filesystem Hadoop with the Hbase database on top as storage backend. This allows OpenTSDB to provide high performance in writing and reading the TS data. Moreover by using Hadoop with Hbase as storage backend OpenTSDB can provide replication and is able to scale the storage backend. However it is not possible to change the storage backend because OpenTSDB's architecture is tightly coupled with Hadoop and Hbase. Therefore the setup of an OpenTSDB instance is not a trivial task regarding the own complexity of the heavyweight Hadoop and Hbase components.

Like KairosDB OpenTSDB does not offer the possibility to create aggregations in an automated way. Moreover it is not possible to combine multiple aggregators.

[11] `https://github.com/OpenTSDB/opentsdb`

3.2.4 InfluxDB

InfluxDB is an emerging TSDB which actually constitutes a TS, metrics and analytics DB. It has been developed in Go with no other external dependencies. It exhibits some interesting features which also overlap with respect to the ones offered by KairosDB.

Similarly to KairosDB, InfluxDB offers a SQL like query language which is however more powerful. This query language enables the filtering via time and any other columns which have been associated to a TS, the selection of multiple series either in a direct comma-based way or via a regular expression, the grouping via time buckets, the filling of missing values for specific time intervals, and the merging as well as the joining of TS. Apart from basic SQL-like querying functionality, the offered language enables the listing of TS as well as the deletion of some TS rows or whole TS. InfluxDB also offers an HTTP(s) API, and offers retention policies for TS data.

Moreover, InfluxDB supports pre-aggregation of data in the form of continuous queries. This is actually one of the most important features of InfluxDB as it allows the aggregation of information in the background in an automated manner without serious implications in the user code as in the case of KairosDB. Through this feature, all required aggregation levels can be supported as each composite metric instance can be considered to be mapped in one continuous query. In this sense, the values of this metric instance are computed automatically and the users just have to write queries based on the name of the metric instance in order to retrieve the respective computations. The sole limitation here is that aggregation is limited to two metrics (instances). This means that if the aggregation over a greater number of metrics needs to be supported, then depending on the aggregation formula, not only one but a set of intermediate continuous queries needs to be generated based on which the final continuous query can be generated and exploited.

Furthermore, InfluxDB supports the merging of time series which are identified via a pattern-based/regular expression, and can deal with distributed settings through shard spaces. This enables the replication of TS data through the specification of a particular replication factor.

A web-based UI is offered by InfluxDB for exploration of the TS data and the production of respective graphs. However, the main difference here is that the InfluxDB UI offers in addition the execution of administration tasks. Also, two dashboards can be integrated with InfluxDB, namely Grafana and Influga.

Older versions of InfluxDB did support multiple underlying storage engines while the recently released, first production-ready version 0.9.0 only supports the key-value store BoltDB.

Concerning the aggregation functions supported, a significantly greater variety compared to OpenTSDB and even KairosDB is offered including: count

(counts the number of TS data indicated by the query), min and max, mean, mode (it calculates the most frequent value in a specific subset of TS identified by the query), median (it identifies the median value in a specific subset of TS identified by the query), distinct (returns only the distinct value for a specific subset of TS identified percentile), histogram (returns the histogram for a specific subset of TS identified which is output in the form of two columns: *(a)* bucket start indicating the starting value in the bucket and *(b)* count indicating the number of values in the bucket), derivative (calculates the derivative of a specific subset of TS), sum, standard deviation, first/last (outputs the first/last point for a specific TS by interval), difference (outputs the difference between the last and first value for a specific TS by interval) and top/bottom (the top/bottom N results by interval for a specific TS).

3.2.5 Selection of TSDB

Table 3.2: Details of considered times series databases

Name	KairosDB	OpenTSDB	InfluxDB
Version	1.0.0	2.1.0	0.9.0
Datastore	H2/Cassandra	HBase	BoltDB
Distributed	no/yes	yes	yes
InMemory	yes/no	no	yes

Table 3.2 presents a basic comparison of established TSDB implementations [12] and their properties. The results are intermediate as our evaluation is ongoing (cf. Section 3.2.6).

The relevant details of the TSDBs are its maturity, available datastores, support of distribution and in-memory storage. The TSDB should be in some mature state in order to provide a stable version, client libraries and an available documentation. Following the strategy exposed in Section 3.2.1 the datastores shall be lightweight and ideally support an in-memory mode. Also they have to offer a distributed architecture to ensure horizontal scaling and replication.

OpenTSDB offers the best maturity regarding the version number. The underlying datastore HBase supports distribution but regarding the architecture of HBase [11] an in-memory mode is missing. Also, it is not a lightweight datastore [12] and an automated set-up as required in our scenario is not a trivial task and hard to script. Consequently, OpenTSDB is not an applicable solution.

From its capabilities InfluxDB seems suited for the outlined approach. Yet, the recently released version 0.9.0 comes with extensive changes in the storage architecture and API design compared to 0.8.0[12].

KairosDB also provides a mature version 1.0.0. It supports the single-site, in-memory datastore H2 and the distributed Cassandra datastore supporting scalability to a hundreds of instances [14]. While Cassandra's resource usage can be limited, in-memory storage is only supported in the commercial version[13].

Following this comparison KairosDB is currently the most appropriate TSDB to use in CLOUDIATOR and hence, the Executionware. The decision has been based on maturity, distribution and the possibility to limit the resource consumption of Cassandra. Nevertheless, the architecture has been designed such that the decision for a TSDB is not tightly coupled to the rest of the system so that the TSDB can be replaced without causing any further issues.

In overall, we state that InfluxDB seems more powerful than KairosDB with additional functionality offered, a more expressive query languages and more configurable ways to distribute and replicate the underlying stores. However, the young age of InfluxDB with frequently changing APIs and feature sets constitutes a too high risk for the success of the project.

3.2.6 Time-series Database Installation

The current version of CLOUDIATOR installs an instance of KairosDB with a Cassandra datastore on each started virtual machine. Cassandra is configured to use only a low portion of a VM's resources to keep the impact on the components running on that VM small.

Whenever possible, we reserve a small fraction of each virtual machine (e.g. 10%) for buffering monitoring data. This strategy assumes that the amount of monitoring data increases linearly with the number of virtual machines. At the same time using a cluster avoids that the TSDB becomes a bottleneck when scaling the application. The reserved area is split into a local storage area and a shared area for replicas of data items created on other virtual machines on the same cloud.

In the long run, we envision that each data element added to KairosDB is stored in a local Couchbase[14] instance using the in-memory memcached option representing the local area and in a distributed Cassandra datastore [14] representing the shared area. Figure 3.1 shows the set-up of the underlying storage systems. It is noteworthy that each cloud uses its own distributed storage. This

[12] `https://influxdb.com/docs/v0.9/introduction/overview.html`
[13] `http://www.datastax.com/`
[14] `http://www.couchbase.com/`

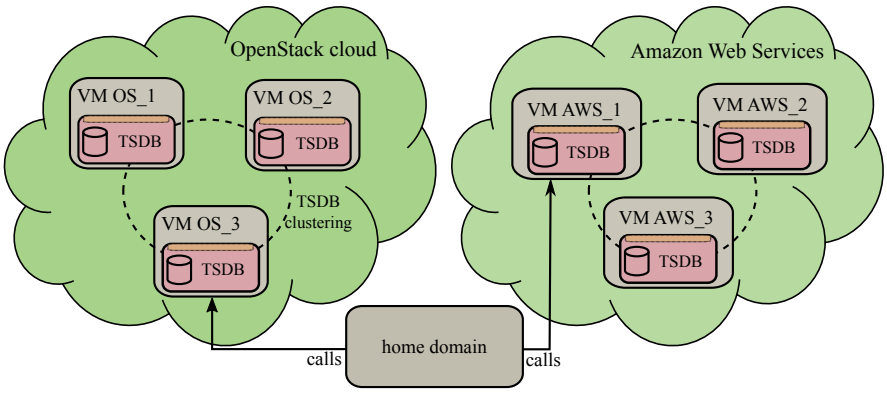

Figure 3.1: TSDB-based architecture of CLOUDIATOR with the local areas marked orange

set-up avoids that the storage suffers from large latencies and that additional costs incur for inter-cloud traffic.

3.3 AXE

From a CLOUDIATOR point of view, there are two scenarios where aggregation on monitoring data is needed: *(i)* A user has requested that monitoring data be collected and aggregated such that it is available outside CLOUDIATOR. *(ii)* Scalability rules require that data to be aggregated to build higher level metrics on which rule conditions can be checked against. In both cases, the AXE component uses the same chain of mechanisms to provision the requested information. The expressiveness of the Scalability Rules Language provides an upper bound on the complexity of queries, aggregation rules, and scaling actions that have to be supported by AXE.

3.3.1 Aggregation

Section 3.1 clarified when and how monitoring data is collected by VISOR: *(i)* ensure that all data needed for aggregation is available, *(ii)* define where the aggregation is performed, and *(iii)* specify where to put the results. The basic concept we use is that the monitor agent forwards the data to a TSDB instance running on the same virtual machine. The latter is responsible for making the monitoring data available to the aggregation functionality including relaying the data to multiple locations if necessary. The aggregation functionality is implemented by aggregation processors.

We distinguish between three scopes that define where the aggregator shall be run: The *host* scope considers aggregation tasks that take into account only

measurements from a single virtual machine. In that case, the collector will forward the data to a virtual machine-local aggregator to do the aggregation. Afterwards, the aggregator will relay the resulting data to the collector again. The *cloud scope* deals with data from multiple component instances or virtual machines from within one cloud. Finally, the *cross-cloud scope* defines aggregation on data from different clouds. For that scope, aggregation happens in the home domain. We use a dedicated collector for each of the scopes.

Aggregations with a cloud scope are triggered from the home domain (*i.e.*, they run within COLOSSEUM). Yet, as the aggregation takes place using the Cassandra store, only the results are sent to the home domain. The results of such a process are stored back to the cloud using any KairosDB instance and the shared area. In case this data has to be made available to the user, a further aggregator is running that pulls this data from any of the cloud's KairosDB instances and stores it in the KairosDB instance running in the home domain. Finally, aggregations in the cross-cloud scope are also run through an aggregator in the home domain, but then stored in the KairosDB of the home domain. Higher-level metrics working on this data will read from the home domains KairosDB and store results back there.

(Aggregated) monitoring data whose collection was requested by the user is always stored in a TSDB in the home domain. It is accessible via the COLOSSEUM API. Improvements on aggregator locations are subject to ongoing work.

3.3.2 Scalability Rules Language

Adaptation describes the capability of the application to autonomously evolve under changing conditions such as varying system load. This may be needed when more users access the application instance than anticipated by the application owner. In a cloud environment, the most frequent reaction to such events will be a scale out/in of individual components or groups of components or the scale up/down of individual virtual machines or groups of virtual machines. For that reason, AXE supports an auto-scaling functionality based on the *Scalability Rules Language (SRL)* [10, 13]. In order to realise this functionality, AXE enables the specification of (hierarchical) metrics, conditions on these metrics, and actions to be executed when the metric conditions are fulfilled. It is important to note that these rules do not have to be provided with the deployment description, but can be added and changed while an application instance is running.

The SRL [10] is a provider-agnostic description language. It provides expressions to define the monitoring of raw metric values from VMs and component instances and also mechanisms to compose higher-level metrics from raw metrics. Moreover, it comprises mechanisms to express events and event patterns on metrics and metric values. Finally, SRL captures thresholds on the events and

actions to be executed when thresholds are violated. A simple SRL rule in prose may be: *add a new instance (scale-out) of this distributed database if* (i) *all instances have a 5 minute average CPU load* $> 60\%$*,* (ii) *at least one instance has a 1 minute average CPU load* $> 85\%$*, and* (iii) *the total number of instances is* < 6*.*

3.3.3 Auto-Scaling

AXE's auto-scaling capabilities for individual components of an application instance require the aggregation of metrics and the evaluation of conditions on these metrics. The generation of monitoring data and their aggregation has been discussed earlier in this section.

In order to evaluate the conditions on the metrics, we apply the strategy to consider conditions on metrics as binary metrics by themselves. These metrics take values in $\{0, 1\}$ and their value is computed as a function that compares the values of the source metric against the threshold of the condition. Composite conditions are computed from their source conditions.

The SRL-engine is an aggregator-like sub-component of AXE that runs in the home domain. Its sole task is to check for all conditions that cause a SRL-related action whether the conditions are satisfied. In case it is, the SRL-engine triggers the actual actions at the Deployment Engine. In addition, it stores the fact that this action has taken place as a separate metric value in the KairosDB instance in the home domain so that it can be queried by higher-level components. Whenever the application instance has changed either by manual intervention or by running a scaling action, the SRL-engine adapts the scalability-related configuration if necessary. This is for instance the case whenever the abstract rule description requires conditions to take into account all instances of a particular component.

3.4 `MetricsCollector`

This section clarifies the functionality provided by the `MetricsCollector` component and how it can be exploited in order to realise a monitoring system within PaaSage. It is organized into four main sub-sections which attempt to highlight: (a) the two main modes of operation of the `MetricsCollector`; (b) the main assumptions imposed by `MetricsCollector` on the environment in order to properly function as expected; (c) the main `MetricsCollector` architecture along with the most important components; (d) the main modes of interaction with the `MetricsCollector`.

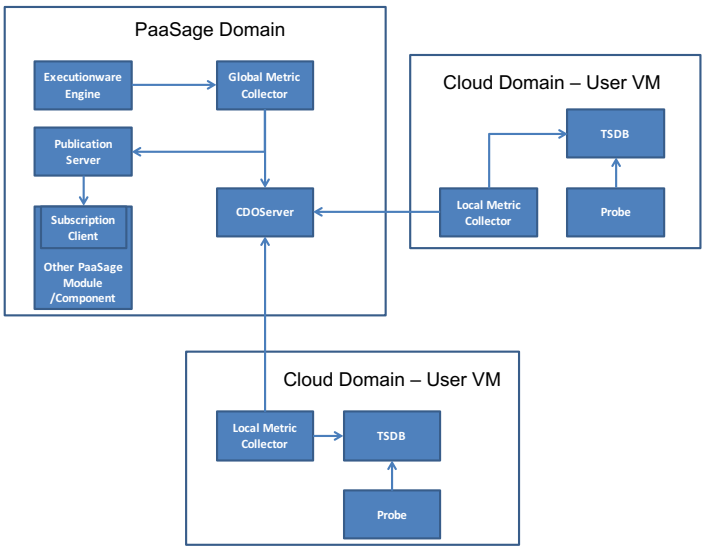

Figure 3.2: The external architecture of the monitoring system

3.4.1 `MetricsCollector` Modes of Operation

The `MetricsCollector` implementation provides two different run time modes: *global* and *local*. These two modes of operation can be clearly seen in the external architecture of the monitoring system depicted in Figure 3.2 which also indicates the relation of both `MetricsCollector`types to other PaaSage components.

The global `MetricsCollector` is deployed in the PaaSage operator domain and is responsible for the aggregation of global, non-single-cloud based metrics based on values of cloud-specific metrics (produced by local `MetricsCollectors`) as well as their storage. The measurements produced are associated to the current execution context of the corresponding application instance. Both the reading and the storage of metrics is performed via the CDO (metadata database) server. The global `MetricsCollector` is also responsible for informing interested components for fresh metric measurements via a publish-subscribe mechanism based on ZeroMQ.

On the other hand, a local `MetricsCollector` is deployed on each application VM of the current application instance and is responsible for the measurement of the respective cloud-specific composite metrics involved (i.e., resource or software component metrics mapping to the VM or the components it hosts, respectively). The aggregation relies on raw measurements produced by probes and stored in the local TSDB. The generated measurements, apart from being stored back in the local TSDB, may be also stored in the CDO Repository.

The decision on the latter depends on the following two cases: (a) we have a top-level metric for which subscribers exist (including the Executionware which has to detect violations of SLOs as well as generate events which might trigger scalability rules) and (b) we have metrics which are children of a global metric, i.e., these metrics are used to compute the global metric—in this latter case we need to store their measurements in CDO such that the global `MetricsCollector` can read them and perform the respective global metric aggregations.

3.4.2 Main Assumptions

The main assumptions of this software component is that Upperware components have left a correct CAMEL (deployment, execution and metric) model in the CDO server of the PaaSage platform. In particular, it assumes the existence of an execution context, the specification of necessary metric instances to be monitored, and the component instances to be measured. Furthermore, a `MetricsCollector` instance operated in local mode assumes that a KairosDB is running and that necessary location and log-in credentials have been provided to it. The global `MetricsCollector` instance provides a publish-subscribe interface through which aggregated measurements are propagated to interested components (e.g., the *Reasoner*). Both types/modes of `MetricsCollector` operation require an appropriate connection to a CDO server (which should contain the CAMEL model of the application to be monitored) such that the necessary information concerning the metric computations are fetched and that the aggregated measurements produced are stored in this server.

3.4.3 Architecture

Both a local and a global `MetricsCollector` are instances of the same class, named `MetricCollector`. This means that both the local and global measurement functionality has been encapsulated in the same code in terms of the same Java class. Figure 3.3 depicts the internal architectural of the `MetricsCollector`.

The core functionality of any `MetricsCollector` comprises an interface with three main methods: (a) *readMetrics* – used for reading the specification of the metric instances that the `MetricsCollector` is in charge of handing and setting up the aggregation code, (b) *updateMetrics* – used for reading the specification of those metric instances that have been updated and then updating the respective aggregation code and (c) *deleteMetrics* – used to delete all aggregation code for all metric instances pertaining to a specific execution context given as input.

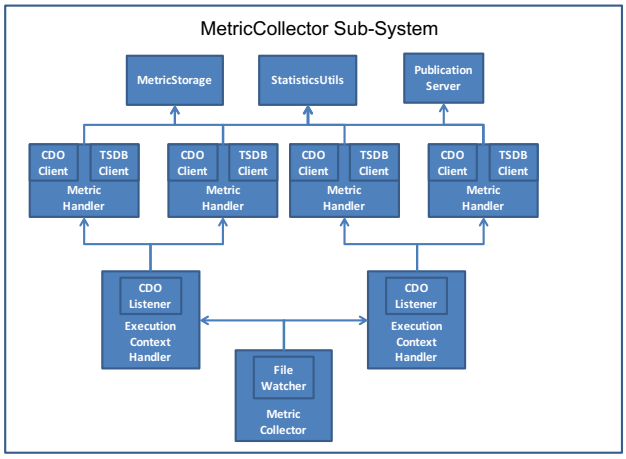

Figure 3.3: The internal architecture of the monitoring system

A global `MetricsCollector` comprises two optional components: the *CDOListener* and the *PublicationServer* (currently realized based on the ZeroMQ framework). Through these two components and the respective `MetricsCollector` aggregation code, the publication of top-level metric measurements is supported to interested subscribers (e.g., PaaSage components). The actual publication of a measurement is supported via calling the respective *submitValue* method of the PublicationServer. The *CDOListener* is responsible for detecting measurements produced for particular (local/cloud-specific) top-level metric instances and publishing them via the *PublicationServer*. The publication responsibilities are completed through the `MetricsCollector` aggregation code which generates measurements for global top-level metrics and publishes them again via the *PublicationServer*. To receive values, an interested component has to just customize the run method of the *SubscriptionClient* class in order to be able to exploit the measurement retrieved for its own purposes.

Execution Context Handling

It must be highlighted here that metric instances to be handled are always associated to a specific execution context. This not only makes their handling easier (as they are grouped in a specific container) but also enables to create a separate functionality in terms of a thread which will be responsible for the setting up and management of the aggregation functionality for each metric in the group. This thread is actually an instance of the class *ExecutionContextHandler* and has similar functionality with respect to `MetricsCollector` but it is restrained in the form of only a metric group. Apart from the metric group handed-over by `MetricsCollector` to the *ExecutionContextHandler*, the latter component

enhances this group by retrieving the specification of other composite metrics at lower levels which are needed for the computation of the metrics in the initial input group. This is performed carefully such that when a component metric is used for the computation of two original metrics, then this component metric is included just once in the group. This enables to avoid having duplication of measurement effort. In the end, after the final metric group is constructed, one *MetricHandler* thread is created for each metric in the group.

Obviously, the careful updating of the initial group is addressed not only in the initial case but also in the case where metric instances in the group are updated as this can raise the need of removing or adding metric instances in the *ExecutionContextHandler*'s metric group.

Metric Handling

A *MetricHandler* constitutes the basic, elementary piece of functionality in terms of a thread devoted to the computation of measurements for a metric instance. The frequency and window of computation is calculated based on the information stored in the CDORepository for the metric instance handled. The MetricHandler thread interfaces with a TSDB client or a CDOClient, depending on whether it is launched in the context of a local or a global `MetricsCollector`, in order to perform the aggregations. In the former case, the TSDB client is used in order to perform aggregated queries to produce the respective measurements of the metric instance handled. In the latter case, the CDOClient is used to either perform aggregated queries over measurements in the CDO Repository or to collect the appropriate measurements and then perform the aggregation over them (in case a particular aggregation function is missing from those supported by CDO/Hibernate Query Language). In both cases, the same client is used to store back the respective measurements with the sole exception that a local *MetricHandler* might need to also store a measurement value in the CDO Repository (for a local metric instance required for computing a global one). In case of top-level global metrics, a global *MetricHandler* will also exploit the *PublicationServer* in order to publish the respective measurements to the interested subscribers.

Aggregation and Database Access

Database Access When measurements or respective generated (scalability rule-based) single event instances need to be stored in the CDO Repository, then the `MetricStorage` class can be utilized which provides an interface with two respective methods: (a) *storeMeasurement* and (b) *storeEvent*. In the first method, you need to provide the measurement value produced, the *CDOID*s of

the corresponding metric instance, of the current execution context and of the object measured (e.g., *InternalComponentInstance* or *VMInstance*) as well as the *MeasurementType* mapping to the respective types of measurements that are supported by PaaSage (application, component, resource and resource coupling measurement types). Once this input is provided, the respective code stores the measurement in the appropriate execution model as well as associates it to the appropriate objects (also given as input – e.g., execution context).

For the second method, it is necessary to specify the event status, the *CDOID*s of the measurement and of the single event to be instantiated as well as optionally the layer on which the event/measurement applies. The respective code, once the input is provided, takes care of generating the event instance, associating it to the respective objects as well as storing it in the appropriate model. Additionally, it checks whether SLOs have been defined for the respective metric condition. If this is the case, then it creates an *SLOAssessment* instance with the appropriate assessment result which is stored in the current execution model as well as associated to the current execution context.

Aggregation Concerning the aggregated measurement production, the *StatisticsUtils* class encompasses methods which cover the missing aggregation functionality of the selected TSDB or the CDO Repository. The aggregation methods additionally supported are: (a) *mode* calculating the most frequent value of a set of measurements, (b) *median* calculating the median value of a set of measurements and (c) *percentile* which calculates the value for which a specific percentage of measurements from the given set is equal or less than it. Each method has been realized to work for measurements produced both from TSDB queries or CDO queries.

There are also two specialized classes which can be regarded as those which drive the aggregation for a particular metric. The *AggregationInfo* class stores important information which is needed for the aggregation of measurements of a specific metric instance, including the metric function involved, its arity and arguments as well as the main aggregation period to be considered. Such information is then exploited by extensions of the *AggregationNode* class which serve as drivers to specific types of TSDBs. Currently, both the KairosDB and the InfluxDB TSDBs are supported and the respective implementations of the *KairosAggregationNode* and *InfluxAggregationNode* have been generated.

An *AggregationNode* comprises the aggregation info, the mode of aggregation, a CDOClient and a specific value. The latter value is actually used to represent constant values and not actual metrics, so in this case the aggregation info is not given. Depending on the type of TSDB supported, the respective extension of *AggregationNode* comprises also the reference to a client which can be used to store measurements as well as perform aggregated or normal queries

over the respective TSDB. Through all this information and internal component references, the aggregation can then proceed based on the TSDB-type-specific-logic encapsulated in the *calculate* method. This method, when called, returns the aggregated measurement value to be stored in the respective places (TSDB, CDO Repository) by the corresponding MetricHandler method.

Both an aggregation node as well as its encompassed aggregation information is produced for a particular metric instance through calling the *setupAggregation* method which again contains some TSDB-type-specific logic needed in order to create specific parts of the required aggregated information. In the case of KairosDB, this method creates the respective aggregator (instance of *Aggregator* class) which has to be employed in order to execute an aggregated query on KairosDB. In the case of InfluxDB, this method creates continuous queries which map metric instance names to particular tables. In this way, the retrieval of aggregation information in this latter case is based on simple queries which retrieve the latest value of the respective table. Thus, there is no need to perform an aggregated query as the results are produced in the background based on the definition of the respective continuous query for the particular metric instance at hand.

Other Utilities and Integration

The `CDOUtils` component provides assisting methods which can be exploited to fulfill different types of functionality. In particular, the following methods have been realized: *(i)* `canPush`: examines whether a particular metric instance is a top-level one such that it has to be published to subscribers via the PublicationServer; *(ii)* `getTopMetrics`: it is used to obtain all instances of top-metric levels that have to be measured; *(iii)* `getGlobalMetrics`: from a list of top-level metric instances, get those which are global and have to be measured by a global `MetricsCollector`.

The last two methods could be used, for example, by AXE in order to delegate the measurement of metric instances to local and global `MetricsCollector`. The global top-level metrics will be delegated to the global `MetricsCollector` while the local top-level metrics will be delegated to the local `MetricsCollector` on which they apply (e.g., based on the component or VM instances for which they measure the respective particular property). Obviously, some additional logic is included in the latter case but it is quite easy to implement as each metric instance is associated to a specific object binding which can be followed in order to discover the particular user VM for which the metric instance must be delegated to the corresponding local `MetricsCollector`.

3.4.4 Modes of Interaction

A `MetricsCollector` can be exploited either by directly calling its main methods or indirectly by updating a particular file. In the second case, an instance of the *FileWatcher* class can be used for indirectly interacting with `MetricsCollector` in order to execute one of its methods. In particular, such an instance monitors the content of a particular file and when updated, it reads it and invokes the respective `MetricsCollector` method reflected by the file content along with the respective input parameters also contained in this file.

3.4.5 Integration with AXE

AXE exploits the functionality of the `MetricsCollector` running in global mode to relay its collected and aggregated metrics to the meta-data database (CDO) and to inform other interested components. For this purposes it enhances the metric collector by a remote interface (RMI) allowing remote access to its functionality.

— 4 —

Further Aspects

This chapter captures aspects that so fare have not been presented. We start with a description of the testbeds in Section 4.1.

4.1 Testbeds

For the development and testing of the current prototype, two testbeds are being provided by the consortium. GWDG and FLEX run OpenStack and Flexiant Cloud Orchestrator (FCO) respectively. The following paragraphs provide an overview of the size and power of the respective testbeds.

4.1.1 GWDG's OpenStack Testbed

GWDG provides its in-house *GWDG Compute Cloud* with virtual machine images including various Linux operating systems pre-configured for instant use. To accommodate diverse requirements of its users, in Y3, GWDG extended its VM flavour offerings to 17 and operating system selection to 13, which are regularly updated to offer newer, safer and more stable releases. Using these images, users can create virtual machines and perform operations on them including suspending, pausing, rebooting, snapshotting, and deleting. The platform further allows assigning public IP addresses to virtual machines and to define firewall rules. The GWDG Cloud is based on the Grizzly release of OpenStack. This can be accessed through the web interface of GWDG portal or directly using the OpenStack API.

In Y3, GWDG extended the infrastructure underlying its compute cloud. The hardware underlying *GWDG Compute Cloud* now includes 38 nodes, having 152 processors, 2432 cores, and 9728 GB of memory. The performance metrics include an overall Linpack value of 18.039 GFLOPS, an overall STREAM value of 4.505 G/s and energy efficiency of 533 GFLOPS/KW.

The *GWDG Compute Cloud* is used for development, testing and deployment purposes. In addition GWDG provides support for resolving any access or deployment issues.

The LSY flight scheduling prototype features a Read-Write intensive, high throughput and low latency application. The *GWDG Compute Cloud* satisfies these requirements. Similarly, the ASCS application requires sharing of NFS file system for coordinating simulation load among a growing number of VMs. GWDG supports AGH, LSY, ASCS or other partners in meeting server side requirements and resolving any network, storage or infrastructural issues faced.

So far, the Year-3 use of the GWDG cloud testbed by the consortium adds up to 30 VMs, 134 vCPUS, 162 GB memory and 29 public IPs, with approximately 1420 GB disk space.

Further more, GWDG is aiming at providing a basic analytics component so that future values of service level parameters may be foreseen to avoid potential violations.

All machines are operated in the data centre of GWDG which is provided in accordance with European data protection and privacy regulations.

4.1.2 Flexiant Testbed (FLEX)

Flexiant provides a constantly evolving cloud virtualisation environment for the PaaSage consortium based on the Flexiant Cloud Orchestrator (FCO). An FCO environment enables customers to deploy virtual machines (VMs).

This testbed has been extended since M18 [6] with respect to functionality, scale, and performance. Here, we describe recent changes for improving both the testbed itself and integration with PaaSage.

Integration Mechanism

A RESTful interface was developed in FCO and released to v4.2 of FCO. Also support was added for FCO to interface with the DASEIN Cloud API aggregator[1].

Triggers

A cost related FCO Trigger has been created for translating Flexiant Units to CPU Hours to gauge the rate various assets being charged on Flexiant provided PaaSage platform for both by usage or time. This is for use in PaaSage by the use cases taking advantage of the Utility Function [4].

[1] `https://github.com/greese/dasein-cloud-flexiant`

Table 4.1: FCO Cluster 2 hardware resources

#nodes	cpu/ram
1	8 * Dual-Core AMD Opetron Processor 8820 /RAM 64GB
2	4 * AMD Opetron Processor 6320 /RAM 32GB
2	8 * AMD Opetron Processor 6212 /RAM 64GB
1	16 * Quad-Core AMD Opetron Processor 8356 /RAM 64GB
2	16 * AMD Opteron Processor 6366 HE / RAM 128 GB

Table 4.2: FCO Cluster 1 hardware resources

#nodes	cpu/ram
2	16 * AMD Opteron Processor 6366 HE / RAM 128 GB
1	8 * AMD Opteron Processor 6212 / RAM 64 GB

Hardware

Flexiant added a second cluster PaaSage testbed recently to support the two clusters available in PaaSage. These improvements are detailed in Table 4.1.

To cater for the needs of the PaaSage project a number of improvements have been made to the existing PaaSage testbed, expanding it to include 4 new high performance compute nodes. These nodes are each specked with 128 GB of RAM and 16 Cores, which has resulted in an additional 512 GB of capacity, an almost doubling of the previous capacity.

In addition to this capacity upgrade the backend storage for Cluster 1 has now been updated to use Ceph storage. This has resulted in an increase of speed available to VMs as well as an increase in capacity to the total overall storage for the Cluster.

A further improvement has been the splitting of the FCO management box from the back end database. The database has now been placed onto a new separate node. This allows an increase in the total number of API requests as well as improved performance for both the database and Web console.

The new platform topology after these improvements is shown in Figures 4.1 and 4.2.

Software

In addition to the updates to the testbed, Flexiant have also updated the cloud software used on the testbed. Currently the testbed is on version 5.05 of FCO, which is the most current version available since September 2015.

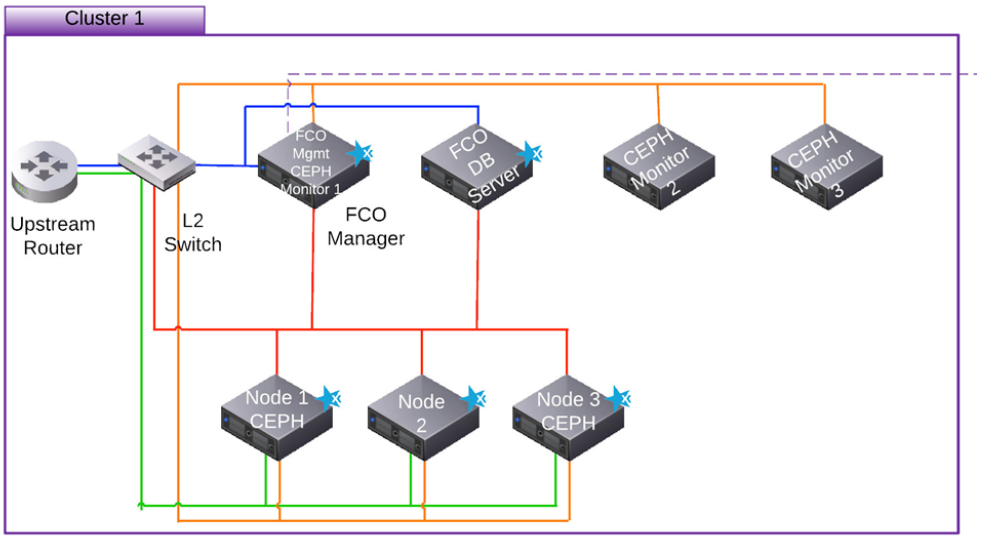

Figure 4.1: FCO Cluster 1 topology

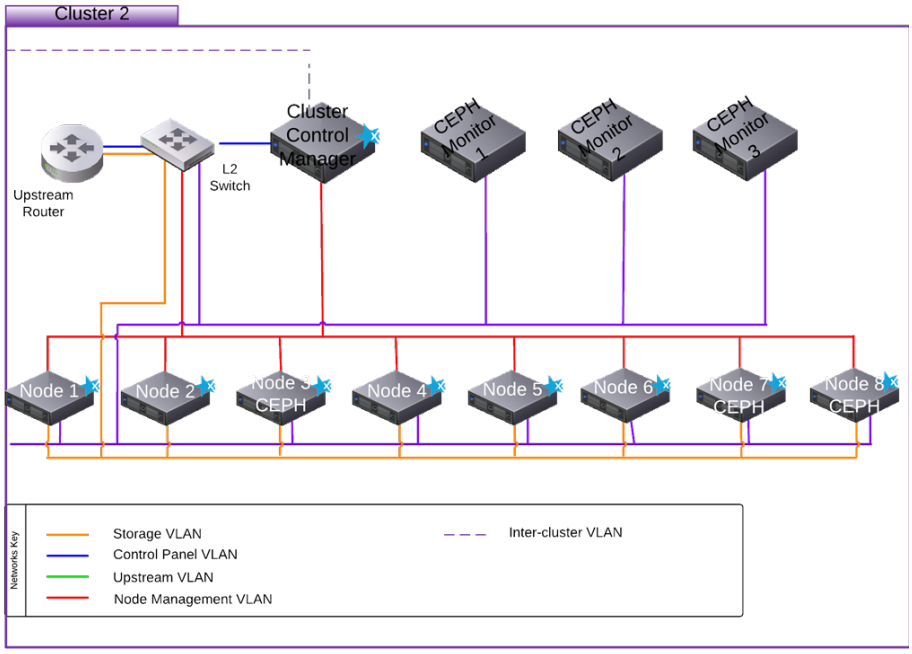

Figure 4.2: FCO Cluster 2 topology

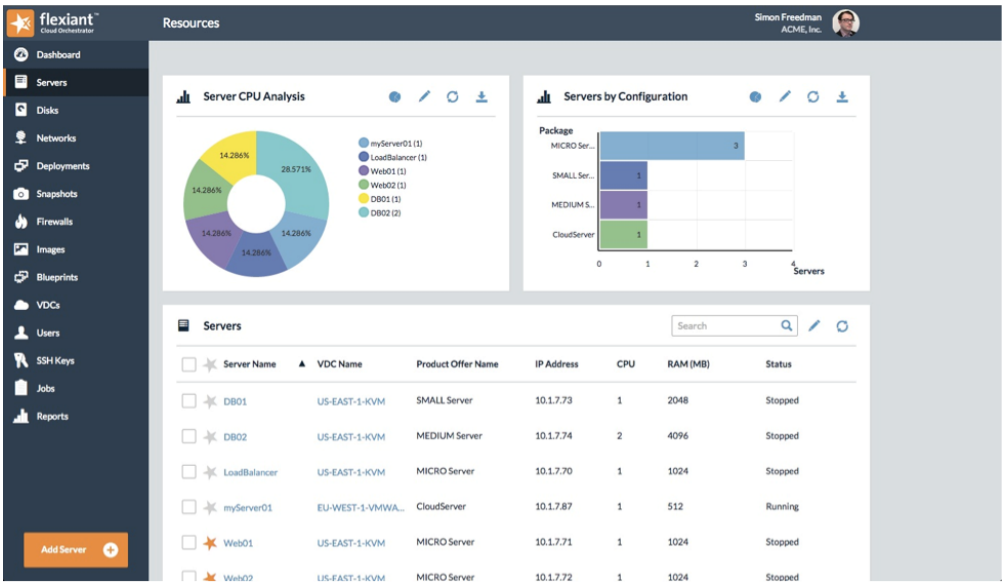

Figure 4.3: User Interface of current FCO version.

The User Interface has been rebuilt and offers increased speed, flexibility, and usability. This improvement resulted in a cleaner more user-friendly experience (*cf.* Figure 4.3).

FCO now supports the addition of multi-tier storage. This can be offered to customers using FCO products and can be limited to a certain value of I/O operations per second (IOPS) or disk throughput. These can be used to create product offers, which either restrict the speed of disks available, or offer higher speed disks. The values available for limitation depend on the hypervisor in use.

4.2 Deployment Controller

Considering the fact that the Executionware already provides a RESTful interface and an associated application server, it is the natural choice to integrate a common controller functionality for the entire PaaSage platform. This feature makes COLOSSEUM (*cf.* Section 2.1) the central entry point in PaaSage for deployment.

In order to achieve this functionality the COLOSSEUM core has been extended with a REST API allowing to send models of applications to the PaaSage platform, to run the Reasoner on those models and possibly to launch the deployment of the application. Third-party and other PaaSage components such as the social network [5] can use this API in order to trigger deployments of

Figure 4.4: Environment for deployment testing

CAMEL applications. Indeed, an integration with the social network is foreseen for the final version of PaaSage [9]. Also the testing environment provided to the PaaSage consortium (*cf.* Figure 4.4) makes use of this API.

Requests to the deployment API can be processed by COLOSSEUM's asynchronous queuing and execution system. The backend is responsible for launching of components from the PaaSage Upperware and then invoking them. In particular, the API allows loading of CAMEL models into the CDO server as well as triggering individual steps from the Upperware up to deploying the application through COLOSSEUM.

— 5 —

Conclusion and Future Work

The Executionware constitutes a fundamental part of the entire PaaSage system and its architecture. The primary purposes of the Executionware are *(i)* to enact interacting with the cloud providers through their respective and largely inhomogeneous APIs, in order to support the creation, configuration as well as tear down of virtual machines and virtual networks; *(ii)* to enact the deployment of application components such as load balancers, databases, and application servers across the created virtual machines; *(iii)* to support the monitoring of both virtual machines and application component instances by provisioning of appropriate sensors/probes and by supporting a reporting interface for applications; *(iv)* to support the aggregation of raw metrics coming from the probes to higher-level composite metrics, and the evaluation of any of these metrics according to conditions and thresholds; *(v)* to report back metric values to the Upperware and to store them in the Metadata-Database (MDDB).

Throughout the PaaSage project, the Executionware has been designed and developed in order to fulfill the aforementioned tasks. Following the concept of *divide et impera* this has led to a set of components that all support a sub set of the requested features, but whose interplay emerges to the desired functionality. In the following, we re-visit the individual components and by explicating the way to access them and further sketch an outlook on their possible future.

The CLOUDIATOR suite consists of several individual tools that are re-captured in the following. Table 5.1 presents an overview on these components, their license and whether they can be used independently. All of these components are released publicly through their main repository hosted on GitHub. Besides, regular releases will also be available through the PaaSage OW2 repository[1]. The development of COLOSSEUM will be continued in the scope of the CloudSocket[2] project with a focus on fault tolerance and the aim to integrate PaaS-like of-

[1] `https://tuleap.ow2.org/projects/paasage`
[2] `https://cloudsocket.eu/`

component	source	standalone
COLOSSEUM	http://git.io/vn6mI	no
AXE	http://git.io/vn6mW	yes
SWORD	http://git.io/vn6mi	yes
LANCE	http://git.io/vn6m5	yes
VISOR	http://git.io/vn6mA	yes

Table 5.1: Components of the CLOUDIATOR suite (https://github.com/cloudiator) and their download locations. All components have been released under the Apache 2.0 license.

ferings. Summarising, CLOUDIATOR consists of COLOSSEUM, AXE, SWORD, LANCE, and VISOR.

The COLOSSEUM component runs in the domain of the PaaSage operator and represents the central access point for any clients through a REST as well as a Web UI. Clients may either be human operators or other tools including the components of the Upperware. COLOSSEUM internally stores information about cloud providers, created virtual machines, components, and component instances.

SWORD is a library that provides an abstraction layer of the various cloud providers. In particular, it encapsulates the differences between them with respect to terminology and technical differences such as provisioning of floating IPs, passwords, and access to virtual machines. LANCE runs in the cloud domain. In particular, COLOSSEUM will deploy one instance of Lance on each virtual machine it creates. LANCE is responsible for executing the life-cycle of the component instances to be installed on the virtual machine. Hence, LANCE executes the scripts to download, install, configure, and start component instances. Just as LANCE, VISOR runs in the cloud domain and is responsible for collecting monitoring data from virtual machines and component instances. In particular, COLOSSEUM will install an instance of VISOR in any virtual machine it creates. Whenever COLOSSEUM is requested to monitor certain aspects of an application and/or virtual machine, it will connect to the VISOR of that virtual machine and request the installation of a sensor together with an interval.

AXE is a two-purpose component that runs partially in the home and partially in the cloud domain. Its first task is to post-process the data collected by the visor component. In particular, Axe is capable of executing aggregation functions on the monitored data such as computing amongst other averages, medians, and quantiles. It may relay selected metrics to other components including the Upperware.

Bibliography

[1] Ian Clarke, Oskar Sandberg, Brandon Wiley and TheodoreW. Hong. 'Freenet: A Distributed Anonymous Information Storage and Retrieval System'. English. In: *Designing Privacy Enhancing Technologies*. Ed. by Hannes Federrath. Vol. 2009. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 46–66. ISBN: 978-3-540-41724-8. DOI: 10.1007/3-540-44702-4_4.

[2] The PaaSage Consortium. *D1.6.1—Initial Architecture Design*. PaaSage project deliverable. Oct. 2013.

[3] The PaaSage Consortium. *D2.1.3—CLOUDML Implementation Documentation Final version*. PaaSage project deliverable. Sept. 2015.

[4] The PaaSage Consortium. *D3.1.2—Product Upperware*. PaaSage project deliverable. Sept. 2015.

[5] The PaaSage Consortium. *D4.1.2—Product Database and Social Network System*. PaaSage project deliverable. Sept. 2015.

[6] The PaaSage Consortium. *D5.1.1—Prototype Executionware*. PaaSage project deliverable. Mar. 2014.

[7] The PaaSage Consortium. *D6.1.1—Initial Requirements*. PaaSage project deliverable. Mar. 2013.

[8] The PaaSage Consortium. *D6.1.2—Final Requirements*. PaaSage project deliverable. Sept. 2014.

[9] The PaaSage Consortium. *D6.2.3—Modified System*. PaaSage project deliverable. June 2016.

[10] Jörg Domaschka, Kyriakos Kritikos and Alessandro Rossini. 'Towards a Generic Language for Scalability Rules'. In: *Proceedings of CSB 2014: 2^{nd} International Workshop on Cloud Service Brokerage*. 2014 (To Appear).

[11] Lars George. *HBase: The Definitive Guide*. 1st ed. O'Reilly Media, 2011. ISBN: 1449396100.

[12] Thomas Goldschmidt, Anton Jansen, Heiko Koziolek, Jens Doppelhamer and Hongyu Pei Breivold. 'Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes'. In: *2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, June 27 - July 2, 2014*. 2014, pp. 602–609.

[13] Kyriakos Kritikos, Jorg Domaschka and Alessandro Rossini. 'SRL: A Scalability Rule Language for Multi-cloud Environments'. In: *CloudCom, 2014 IEEE 6th International Conference on*. Dec. 2014, pp. 1–9. DOI: 10.1109/CloudCom.2014.170.

[14] Avinash Lakshman and Prashant Malik. 'Cassandra: A Decentralized Structured Storage System'. In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: http://doi.acm.org/10.1145/1773912.1773922.

[15] D. MacEachern and R. Morgan. *SIGAR — System Information Gatherer And Reporter*. 2010.

[16] B. Morin, O. Barais, J. -M Jezequel, F. Fleurey and A. Solberg. 'Models@ Run.time to Support Dynamic Adaptation'. In: *Computer, IEEE* 42.10 (2009), pp. 44–51. ISSN: 0018-9162. DOI: 10.1109/MC.2009.327.