PaaSage

Model Based Cloud Platform Upperware

Deliverables D5.1.1&D5.3.1

Prototype Executionware,
Prototype New Execution Engines

Version: 1.0

D5.1.1/5.3.1

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

Mr Tom Williamson Tel: +33 4 9238 5072 Fax: +33 4 92385011 E-mail: tom.williamson@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:	10th October 2013
Document	
Period covered:	
Deliverable number:	D5.1.1/D5.3.1
Deliverable title	Prototype Executionware/ Prototype New Execution Engines
Contractual Date of Delivery:	31st March 2014 (M18)
Actual Date of Delivery:	31st March 2014
Editor (s):	Jörg Domaschka (UULM)
Author (s):	Anthony Sulistio (USTUTT), Panagiotis Garefalakis (FORTH), Damianos Metalidis (FORTH), Chrysostomos Zeginis (FORTH), Craig Sheridan (FLEX), Kuan Lu (GWDG), Edwin Yaqub (GWDG), Jörg Domaschka (UULM), Bartosz Balis (AGH), Dariusz Król (AGH)
Reviewer (s):	Tom Kirkham (STFC), Geir Horn (UiO)
Participant(s):	Nikos Parlavantzas (INRIA), Michaël Van de Borne (CETIC), Kyriakos Kritikos (FORTH), Loke Johannessen (FLEX), Philipp Wieder (GWDG), Lutz Schubert (UULM), Daniel Baur (UULM), Maciej Malawski (AGH)
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):	66

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. On the one hand, it is responsible for bringing applications to execution that have been modelled and configured in PaaSage's Upperware component. On the other hand, the Executionware is responsible for monitoring the execution of each individual instance of a component—taking features of the respective cloud providers into account.

This document presents an overview and documentation for the first prototype of the PaaSage Executionware implementation. It states the functional scope of that prototype with respect to the architecture. It also introduces technologies the Executionware is built upon and describes the main Executionware modules, namely the Executionware frontend and the Executionware time-series database. The former serves as the primary access point for higher layers of the PaaSage architecture, mainly the Upperware. The latter collects and accumulates monitoring data and feeds it back to the meta-data database.

This document clarifies the implementation status of the M18 prototype: The Executionware is able to receive and process deployment information from the Upperware. It is capable of deploying an application to a single cloud which is either an instance of OpenStack or Flexiant Cloud Orchestrator. The M18 prototype further installs a series of sensors with each deployed application and evaluates monitoring data from a dedicated source, e.g. CPU load according to a pre-defined metric. Finally, the evaluated monitoring information is fed back to the meta-data database.

In addition to defining and describing the basic functionality of the M18 prototype, this document investigates the extendibility of the Executionware with respect to data farming and workflow processing. In particular, it clarifies the motivation for specialised flavours for dedicated Executionware entities and defines the roadmap to their integration into the final version of PaaSage's Executionware.

Intended Audience

This deliverable is a public document intended for readers with some experience with cloud computing and cloud middleware. It presumes 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. It also describes an overview on the implementation status of the Executionware prototype at month M18.

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 11
2 Summary of Executionware Architecture 11
2.1 Executionware Entities of the PaaSage Infrastructure . . 13
2.2 Entities of the Executionware Run-time System 14
2.3 Entities of the Monitoring Infrastructure 14
2.4 Interaction with Other Work Packages 15
2.5 Conclusions 16
3 The Executionware Prototype 17
3.1 Detailed Scope of M18 Prototype 17
3.2 M18 Components 19
3.3 Conclusions 21
4 Technological Background 21
4.1 Cloudify 22
4.2 KairosDB 26
4.3 Provisioning of Testbeds 30
4.4 Conclusions 31
5 Application Deployment and Cloud Management 32
5.1 The Executionware Cloud Registery 32
5.2 Deployment of Applications 35
5.3 Conclusions and Status of Implementation 37
6 Application Monitoring 37
6.1 Integration to With Executionware 38
6.2 Integration with Meta-data Database 41
6.3 Conclusion and Implementation Status 42
7 Workflow Applications 43
7.1 The Case for Workflow Execution in PaaSage 43
7.2 The HyperFlow Platform 44
7.3 Conclusions and Implementation Status 46
8 Data Farming Applications 47
8.1 The Case for Data Farming in PaaSage 47
8.2 The Scalarm Platform 48
8.3 Conclusions and Status of Implementation 50
9 Conclusion and Future Work 51
9.1 Summary of Implementation Status 51
9.2 Roadmap to Final Version 54
References 57
A Executionware Frontend 58
A.1 Implementation Aspects 58
A.2 Cloudify Cloud Configuration 58

A.3 Cloudify Recipes 58
A.4 CAMEL-based API 59
B Time-series Database and Monitoring 63
C Workflow Platform 63
C.1 Provided Cookbook 64
C.2 Usage Example 64

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. We distinguish between cloud-related and application-related terminology.

Cloud Terminology

Cloud platform A cloud platform is the software run by a cloud provider. In particular, this means that it is the platform that defines the API to interact with any cloud provider running this platform. As the platform is something completely passive it does, however, not define contact endpoints (e.g. URIs) to interact with the provider. The OpenStack software suite and Flexiant's Flexiant Cloud Orchestrator (FCO) are examples for cloud platforms.

Cloud instance A cloud instance is an instantiated cloud platform. That means, it defines the cloud provider and the access points (i.e. URIs) to access the services offered by the provider. For instance Amazon EC^2 is an example of a cloud instance. Within the PaaSage consortium examples include OpenStack run by GWDG, and FCO run by Flexiant.

Cloud A cloud is a cloud instance as seen by a particular tenant of that cloud instance. In consequence, a cloud links a cloud instance with the necessary credentials to access the cloud instance such as `username` and `password`. While this may seem like a unusual terminology, it fits seamlessly the requirements and topics addressed by this document.

Application Terminology

(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.

(Software) component A software component is a set of artefacts that can be brought to execution as an operating system process without any dependencies to other components on binary level. It may depend on other components on higher level protocols such as HTTP. Multiple components may require the same artefacts either by sharing (e.g. shared libraries) or by copying (e.g. configuration files).

Application An application is a set of software components that form a functional closure. The same component (e.g. a PostgreSQL database) may appear in several applications.

Application instance An application instance is an incarnation of an application (and hence, all its software components). The same application may be instantiated several times; in that case, the various instances do not share any state or properties.

Component instance A component instance is an instantiation of a software component within an application instance. Component instances do not share any state among each other. They may, however, cooperate with each other. For instance, a distributed database consists of multiple instances of the database component.

1 Introduction and Overview

The Executionware constitutes a fundamental part of the entire PaaSage system and its architecture. Its focus is two-fold: On the one hand, it is responsible for bringing applications to execution that have been modelled and configured in PaaSage's Upperware component. On the other hand, the Executionware is responsible for monitoring the execution of each individual instance of a component—taking features of the respective cloud providers into account.

This document presents an overview and documentation for the first prototype of the PaaSage Executionware implementation. In order to clarify the background, we first summarise the requirements towards and the initial architecture of the Executionware architecture as defined in deliverable D1.6.1 [2] in Section 2.

Section 3 explicitly states the functional scope of the first prototype with respect to the architecture. It also introduces the Cloudify framework that serves as a starting point for the Executionware deployment and monitoring functionality. Additionally, it describes fundamentals of KairosDB, a time-series database that is used for processing and accumulating monitoring data. Finally, Section 3 sketches the software modules developed for the prototype and how they match the Executionware architecture from Section 2.

Afterwards, we introduce the individual software modules in-depth and discuss their respective implementation status with respect to the M18 prototype. Section 5 presents the Executionware frontend that the PaaSage Upperware uses to trigger application deployment and configuration changes. The frontend also provides a Web-based user interface for configuration purposes. Section 6 introduces the implementation and technology used to realise the monitoring functionality of the Executionware and its link to the meta-data database.

Section 7 and Section 8 discuss approaches of how the current architecture may better support the existing HyperFlow workflow engine and the Scalarm data farming framework respectively. Integrating these two platforms enables dedicated support for workflows and data farming in PaaSage. We conclude this document in Section 9 with re-visiting open issues and clarifying the roadmap to the final version of the Executionware in M36. The appendix of this document contains further technical sources of information about the developed software modules.

2 Summary of Executionware Architecture

The PaaSage architecture as described in deliverable D1.6.1 [2] defines the role of the Executionware in the PaaSage application life-cycle. Together with the

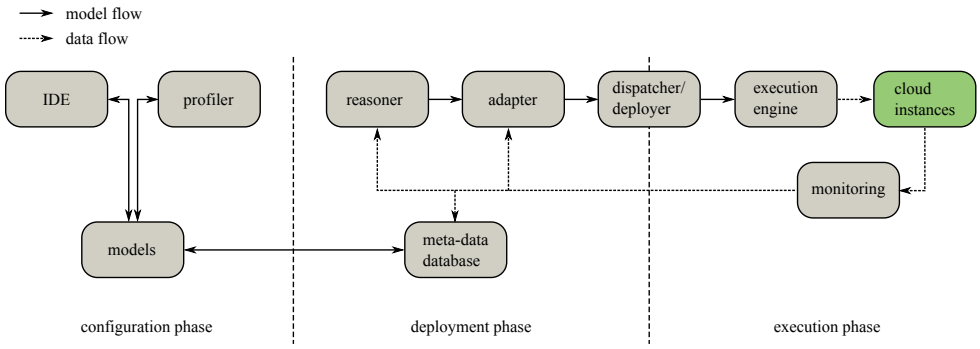

Figure 1: Main PaaSage components and life-cycle direction including data and model flow as deliverable D1.6.1 [2].

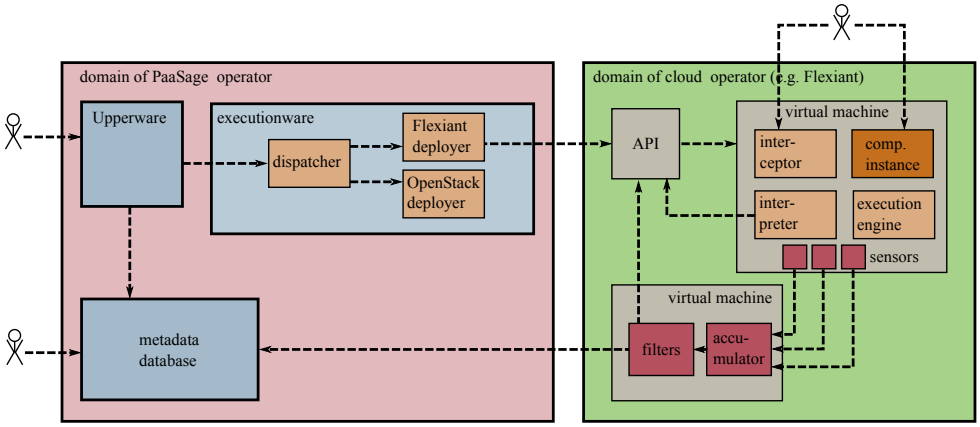

Figure 2: Current architecture of the Executionware together with input to and output from other PaaSage components.

user requirements as defined in deliverable D6.1.1 [6] this tightens the overall architecture of the Executionware. This architecture has already been specified in deliverable D1.6.1 so that this section focuses on recapturing it. Moreover, it identifies its individual logical entities.

Figure 1 sketches the data flow and model flow between the individual PaaSage components at configuration, deployment, and execution phase. It points out that the data flow is triggered by the execution engine in the Executionware and floats back to Upperware components via the monitoring infrastructure and the meta-data database. All components in PaaSage's execution phase on the right hand side of the figure are subject to the Executionware.

Figure 2 shows the architecture derived for the Executionware. As illustrated in the figure, the components of the Passage Executionware can be separated into three groups. The first group forms the static part of the infrastructure that

is hosted by the PaaSage operator and is not necessarily executed in a cloud environment. We present the entities of this group in Section 2.1. The second group of Executionware entities is brought out in the field together with the application's component instances. Their main purpose is the provisioning of a run-time environment for these component instances. The run-time entities are subject to Section 2.2. Finally, information about the behaviour of the deployed component instances has to be collected and further processed. This functionality is realised by the monitoring infrastructure that we introduce in Section 2.3. We conclude this section after a brief sketch of how the Executionware interacts with other components of the PaaSage platform in Section 2.4.

2.1 Executionware Entities of the PaaSage Infrastructure

The PaaSage Executionware provides two types of entities that constitute the static part of the infrastructure.

The Executionware entities that are part of the PaaSage infrastructure run on the premises of the PaaSage owner and not necessarily on a cloud environment (even though they might). The two entity types, Dispatcher and Deployer, offer a deployment service to the components of the Upperware. In particular, the Dispatcher entity is the only access point higher-level entities can use to deploy PaaSage applications.

Dispatcher The Dispatcher is triggered by the Upperware on two occasions. In case the Upperware wants to deploy a new application, the dispatcher receives the information on what artefacts to deploy on which cloud with what configuration and what scripts to run at the various lifecycle events (cf. Section 4.1). In case the deployment of a deployed application shall be changed, the Executionware receives information of what instances to shut down and what application components to start. It is the Upperware's task to ensure the correct order when deploying multiple interdependent application components. The task of the Dispatcher is then to split up that information on a per-cloud and per-component instance and hand it on to the *Deployer* responsible for that respective cloud system. We discuss the API between Upperware and Executionware in Section 5.2.

Deployer A Deployer is capable of transferring the application configuration to a low-level representation that can then be deployed on a cloud. That is, a Deployer processes the data received from the Upperware (and the Dispatcher), turns it into a configuration set-up, and finally deploys this set-up. As such, the implementation of a Deployer may be specific to a cloud platform or to a group of cloud platforms.

2.2 Entities of the Executionware Run-time System

In order for the system to be able to retrieve the necessary information at run time and to act on behalf of the PaaSage system, we need several entities that are co-located with the instantiated components instances. In particular, each component instance is accompanied by the execution engine, the communication interceptor (aka wrapper), and finally, a interpreter entity [2].

Execution engine The execution engine is a service co-located with every component instance. It is not specific to the component, but it may be dependent on the cloud infrastructure it is running on. The purpose of this entity is to provide a communication endpoint under which the component instance (as well as the virtual machine it is executed on) can be controlled and influenced.

Interpreter The interpreter sits in-between the component instance and the cloud infrastructure. As such, it is highly dependent on the platform it is running on, including the operating system. Its main task is to intercept all invocations the component instance targets towards the cloud API; The component instance may do that for multiple reasons: get monitoring data, scale out the application, start sub-processes (as in workflow systems). In order to control the interaction between the component instance and the cloud, the Interpreter appears as the cloud operator to the component instance. That is, it has the same API as the cloud platform. There exist multiple mechanisms to achieve that goal. For instance library overloading, replacement of binaries, re-direction of IP traffic. The concrete technique(s) applied by the PaaSage Executionware will be identified throughout the project. The very same mechanisms can be used to achieve control over other interactions by the component instances such as access to the file system.

Interceptor The interceptor sits in-between external external entities and a component instance. As such, it controls all incoming connections and redirects them to the component instance. This allows, for instance, to count the number of concurrently connected users and open connections.

2.3 Entities of the Monitoring Infrastructure

In addition to run-time entities that shield the component instances and static components that enable the actual deployment of applications, the Executionware captures and processes monitoring data. Virtual machines running component instances are enriched with monitoring probes as needed in order to get

information about the run-time performance of virtual machines, component instance, and hence, about application instances. The additional entities employed by the monitoring infrastructure fulfil the task of accumulating and filtering the monitoring data. In consequence, they help to retrieve useful information from the mere monitoring information.

Sensors Sensors are brought out in the cloud and retrieve monitoring information. This information may either stem directly from the component instances or virtual machines or from monitoring data provided by the cloud provider. Each sensor emits a stream of monitoring data that reaches an accumulator or filter. The necessary sensors are provided as specified by the deployment information passed by the Upperware.

Accumulators Accumulators have the task of combining multiple monitoring streams emitted by various sensors or other accumulators. They may further perform operations on the streams such as computation of average values over certain periods of time. Similar to sensors, an accumulator emits data. Yet, in contrast to sensors, it may also emit more than one data stream towards different sinks.

Filters Filters operate on a per stream basis and search for defined patterns in that stream. Once a pattern is found, one or multiple actions associated with that pattern are triggered as a reaction. Similar to sensors, the necessary patters are application- and component-specific and hence, are passed to the Executionware by the application specification.

2.4 Interaction with Other Work Packages

Within the PaaSage architecture, the Executionware solely interacts with two other PaaSage sub-systems, namely the Upperware and the meta-data database (MDDB). On the one hand, the Dispatcher entity of the Executionware exclusively functions as a sink for the Upperware. That is, it only receives data from the Upperware, but never directly invokes the Upperware. On the other hand, the monitoring infrastructure functions as a data provider for the meta-data database. The Executionware never directly retrieves data from the MDDB.

Input From Executionware

The Executionware receives invocations from the Upperware. In particular, there are three occasions when the Upperware may contact the Executionware:

1. Deploy a new application
2. Change the deployment of an application
3. Stop the execution of an application

Section 3.1 further details the support envisaged for the M18 Prototype. The API between Upperware and Executionware is subject to Section 5.2.

Output to Metadata Database

Besides the fact that monitoring data reaches Executionware filters and may trigger Executionware-specific actions, parts of that data are also relayed to the meta-data database. While on-line filtering enables a quick reaction to abnormal conditions, having data consolidated in the MDDB, enables for off-line data evaluation and analysis. This strategy enables the Upperware to perform queries that are too expensive with respect to performance and time to be executed on-line. Hence, the Upperware can draw much more detailed conclusions regarding the overall deployment quality of an application. In addition to the monitoring data, also events triggered by the Executionware are added to the database. This enables the Upperware to decide on the quality of decisions taken on-line.

2.5 Conclusions

In this section, we have sketched the overall architecture of PaaSage's Executionware. We have seen that the Executionware receives input from the Upperware. As a response to that input, it deploys a new application or changes the deployment of existing applications instances. While an application is deployed, i.e., available for use, all of its component instances are constantly monitored by the Executionware monitoring infrastructure. Executionware accumulators combine the monitoring streams emitted by different component instances. Executionware filters evaluate the monitoring data and react to pre-defined patterns. In addition, the Executionware relays all monitoring data to the meta-data database for later offline evaluation by the Upperware. This closes the PaaSage feedback loop (cf. Figure 1).

While the Executionware architecture and with it this section targets the overall Executionware functionality to be realised throughout the project run-time, the major scope of this document is the implementation status of the Executionware prototype at M18. It is the task of Section 3 to put the Executionware in the scope of the M18 prototype. The section defines the goals for M18 and describes the composition of the logical entities of the architecture in software artefacts. The technical foundation of our implementation is subject to Section 4.

3 The Executionware Prototype

This section builds on the Executionware architecture as defined in Section 2 and clarifies the impact and scope of the M18 prototype. Nevertheless, the content of this section is merely a snapshot of ongoing development work in Work Package 5 and may become outdated as the Executionware evolves towards its final release due at M36.

For the architecture assumes a full-fledged Executionware, we first narrow the scope of functionality envisaged for the M18 prototype (cf. Section 3.1). Then, we define the division of the logical elements of the Executionware architecture from Section 2 into larger software modules (cf. Section 3.2).

3.1 Detailed Scope of M18 Prototype

This section defines the scope of the M18 prototype. In particular, it targets envisaged functionality in terms of general capabilities and covered scenarios. For specifying the capabilities, we follow the structure already known from Section 2 so that we first discuss capabilities of the PaaSage infrastructure, followed by those of the run-time system and of the monitoring infrastructure. The interaction with other PaaSage sub-systems is beyond the scope of this document, as the integration with the other components is subject to M19–M21 (cf. Section 9).

Functionality of the Infrastructure Entities

The primary goal of the Executionware prototype is to provide a solid, yet flexible, foundation for later sophisticated functionality. With respect to application deployment, this means, that at M18, the functionality of the Executionware will correspond to the following items:

1. The Executionware **is** able to receive and process deployment information from the Upperware. Yet, it **only** provides this functionality as long as the deployment targets a single application possibly consisting of multiple components.

2. The Executionware **is** able to deploy this application on a **single** cloud. That is, cross-cloud deployment is not supported at M18.

3. The Executionware **provides** deployers for clouds based on OpenStack and Flexiant Cloud Orchestrator. Deployment may **either** target a Flexiant-based **or** OpenStack-based cloud.

4. The M18 prototype **offers** a Web-based interface to register and manage clouds, accounts, and virtual machine images.

5. The Executionware **provides** mechanisms for the Upperware to derive the right virtual machine image to use for the deployment description. That is, it enables a mapping from the OS description *linux, 64-bit* to image ID *XX-YYYY-ZZZ* provided that the image has been registered earlier.

The requirements regarding the Executionware API and the necessary input data received from the Executionware are further outlined in Section 5.2 and Annex A.4.

Functionality of Runtime Entities

Regarding the runtime functionality, the goal of the prototype is to provide a first step towards the full-fledged implementation. Therefore, our primary focus is on the provisioning of the Execution Engine. In particular, the Execution Engine supported by M18 **is** able to function as a communication endpoint for the infrastructure entities. That is, it **is** be able to support all actions required by the infrastructure components in order to support the deployment of component instances. The realisation of other runtime entities is subject to the M18–M36 period.

Functionality of Monitoring Entities

In general, monitoring serves the purpose of retrieving information about the deployed component instances and further about the resources each instance consumes. Moreover, monitoring includes the retrieval of information about the usage characteristics of particular component instances. This may, for instance, cover how many users are accessing that instance. The M18 prototype provides the following functionality.

1. The prototype **installs** a fixed set of sensors with every deployed component instance.

2. The Executionware **is** able to retrieve basic environment values such as CPU load, memory consumptions, I/O statistics of individual virtual machines and component instances.

The monitoring data obtained by the sensors is processed in two ways. First, all monitoring information is directly filtered by the Executionware in order to identify patterns that require auto-scalability. Second, accumulated monitoring data is redirected to the meta-data database. In particular, the M18 prototype provides the following functionality:

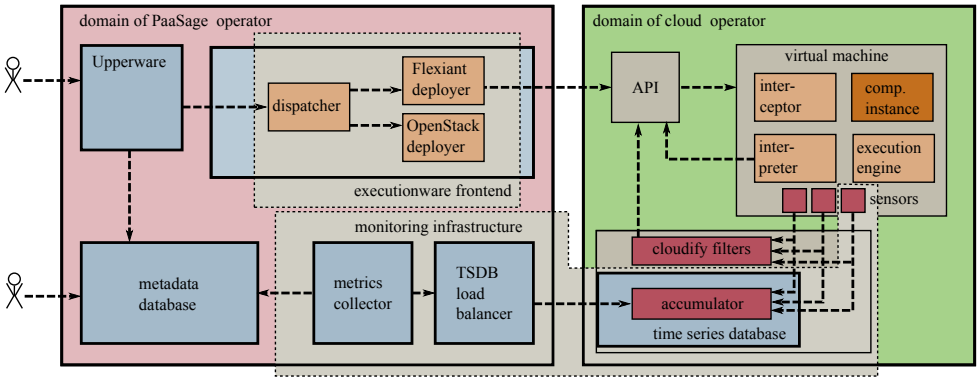

Figure 3: Current architecture of the Executionware and the mapping of entities to software modules

1. The Executionware **is** able to redirect **parts of** all monitoring data to the meta-data database.

2. The Executionware **accumulates** the monitoring data before relaying it to the meta-data database.

3. The Executionware **is** able to trigger the scale-out of a component as the consequence of a hit pattern. The supported patterns **are limited** to a single, pre-defined monitoring source such as CPU. The necessary filtering rule has to be passed by the Upperware together with the deployment information.

4. The Executionware **is** able to combine all monitoring streams of a single cloud in a single location. There reside accumulators, filters, and the linkage to the metadata database.

3.2 M18 Components

The architecture of PaaSage's Executionware as described in Section 2 consists of multiple individual entities. For the implementation of the M18 prototype, we combine multiple of these entities to two larger software modules called Executionware *frontend* and a *time-series database (TSDB)*.

Figure 3 sketches the mapping of individual entities to software modules. In the following, we only briefly describe these modules and describe which entities of the overall architecture they comprise. Both front-end and time-series database are discussed in more detail in Section 5 and Section 6 respectively.

Executionware Frontend

The Executionware frontend constitutes the entry point for the Upperware. It provides a HTTP-based interface the Upperware can use for deploying applications. In consequence, the frontend implements the Dispatcher entity as well as the individual deployers. For the M18 prototype these are a Flexiant deployer and an OpenStack-capable deployer. The deployment itself is realised by rewriting and assembling the data passed from the Upperware to a valid low-level configuration that third party tools can use to execute the actual deployment. The Executionware uses Cloudify (cf. Section 4.1) for that purpose.

The frontend also provides a directory of clouds known to the PaaSage operator. It offers a graphical user interface that the PaaSage operator may use to register existing clouds with the Executionware. The deployment process of the frontend will make use of this information when accessing the cloud. Before a cloud registered with the Executionware may be used for deploying applications, it has to be bootstrapped by the frontend. During that bootstrapping, a time-series database is installed in that particular cloud and interlinked with other time-series databases in other clouds run by the same PaaSage provider.

Monitoring Infrastructure

The main part of the monitoring infrastructure consists of the Executionware time-series database. The Executionware TSDB is a distributed database spanning multiple cloud platforms. It is particularly suited for processing large amounts of streaming data what makes it a perfect match for dealing with the monitoring information issued by all virtual machines and component instances. The individual instances of the TSDB on different clouds are installed and brought up by the frontend whenever a cloud is bootstrapped. It is also the frontend that is responsible for ensuring that TSDBs belonging to the same PaaSage operator find each other and form a unique cross-cloud database.

The TSDB is enhanced by a collector entity that interlinks the time-series database with the meta-data database and further a load balancer entity that distributes queries among the various TSDB instances. While the M18 prototype mainly uses the standard mechanisms to monitoring offered by third party libraries (cf. Section 4) also used for deployment, we also provide some custom sensors for monitoring specific data.

Other Components

Partially, the Executionware prototype builds on existing, software components from third-party libraries. This is in particular true for the Execution Engine that

is not covered by any Executionware modules. This is because the M18 functionality is already covered by the software libraries the Executionware builds upon. The same is true for the filters required to dynamically scale out applications. We discuss all third-party libraries together with other technical background in Section 4.

Apart from Executionware frontend and monitoring infrastructure, the Executionware comprises two add-ons that are concerned with an efficient execution of workflow-oriented and data farming applications. These are subject to Section 7 and Section 8 respectively.

3.3 Conclusions

In this section, we have defined the scope and the capabilities of the M18 prototype. We have further identified key entities that are required to realise this functionality and presented how they are composed to larger software modules. In consequence, the M18 prototype provides a solid technological foundation that enables the development of sophisticated functionality in later stages of the project.

While this section described a high-level view on the Executionware, Section 5 and Section 6 provide a much more detailed and much more technological perspective on the individual software modules. Yet, for being able to distinguish own contributions from background technology, we first present an overview on the technological background in Section 4 that constitutes the foundations of our own implementation.

4 Technological Background

In this section, we present the technological background to our own implementation of the M18 Executionware prototype. On the one hand, this comprises the presentation of third party libraries and tools used by our implementation. In particular, we discuss fundamentals of the Cloudify[1] software that the Executionware uses as its primary deployment tool (cf. Section 4.1). We further introduce KairosDB[2] that the monitoring infrastructure uses as a basis for its TSDB solution (cf Section 4.2). On the other hand, we briefly sketch the testbeds provided by the PaaSage consortium that we use for testing and demonstration purposes (cf. Section 4.3).

[1] `http://www.cloudifysource.org/`
[2] `https://code.google.com/p/kairosdb/`

4.1 Cloudify

Huge parts of the Executionware functionality will be based on the Cloudify open source software framework. The decision for this library was taken for the following reasons: First, it is open-source and its Apache 2.0 license enables adaptation and re-use in both open source and commercial products. Second, the general architecture of the software resembles the architecture defined for the Executionware (cf. Section 2) so that additional functionality can be added and integrated step-wise which enables a smooth implementation kick-off. Finally, Cloudify comes with multiple drivers that support many existing cloud offerings.

For the Cloudify framework already provides many of the functionalities required by the PaaSage Executionware, the Executionware relays functionality to Cloudify wherever possible. Nevertheless, some modifications and extensions are required to the Cloudify configuration mechanism; this is mainly caused by the fact that Cloudify itself is not service-enabled and hence does not support multiple users and remote access to its functionality. Furthermore, it does not support multi-cloud applications. We discuss these issues in Section 5.

In this section, we give a brief overview on the Cloudify software. We start with the general concepts and terminology. Then, we follow the same approach as for the Executionware architecture and consider platform entities, runtime entities, as well as monitoring support. Finally, we turn towards the application model supported by Cloudify and present the lifecycle it enforces for its applications.

Concepts and Terminology

Cloudify distinguishes the main entity terms *cloud*, *service*, and *application*.

Cloud Similar to the definition used in this document a Cloudify cloud maps to a cloud instance combined with access credentials. A Cloudify cloud is defined by providing one or multiple configuration files in the Cloudify configuration directory. This directory is then accessed by the initialisation scripts (mainly the Cloudify shell) when starting and stopping a Cloudify cloud. The start of such a cloud always results in the provisioning of one or more *management virtual machine(s)* on the Cloudify cloud.

Management VM A management VM is a central anchor in a cloud. It provides a Web-based GUI from which the user can see started virtual machines as well as their monitoring data. Using the GUI, the user can scale (expand) applications or services. Further, the management VM collects monitoring data and stores information about an application's state. A management VM also offers a

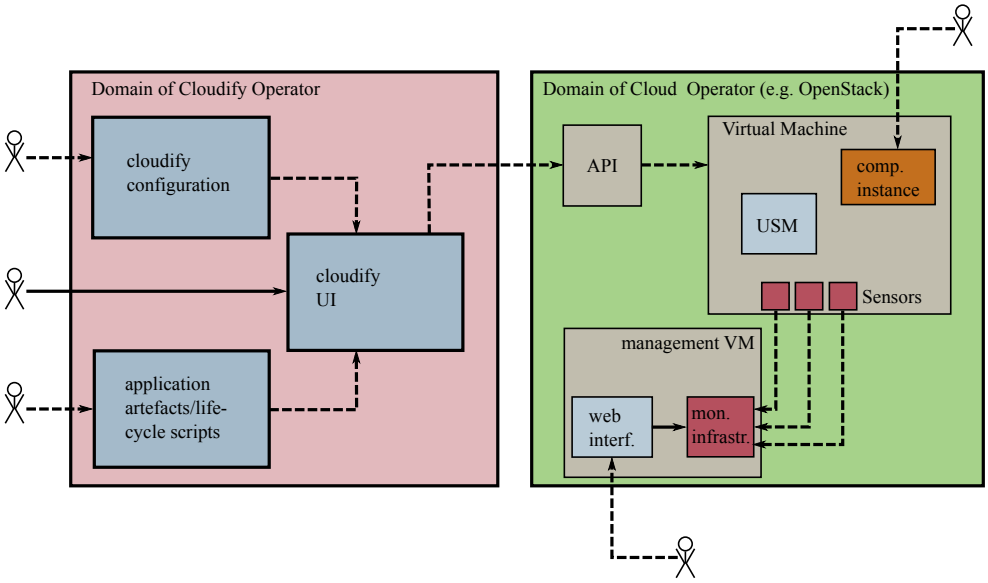

Figure 4: A schematic overview of the Cloudify run-time set-up. The system consists of a file-based cloud configuration repository, management VMs on a per-cloud basis, and deployed applications consisting of multiple services which again consists of multiple instances. All instances of a single service run in a single cloud. Furthermore, the management VM collects all monitoring data issued by the application instances.

REST-based interface that supports the same functionality as the GUI. Amongst others, it enables the deployment and scaling of applications.

Application A Cloudify application is a collection of services and their interdependencies. In contrast to PaaSage applications, Cloudify applications are deployed on **a single** cloud and do not span multiple clouds.

Service A service is the set of instances of a particular binary (such as for instance a Web server). A service may come with several scripts to be executed at particular points in time during the application lifecycle.

Platform Entities

Figure 4 sketches the architecture of a Cloudify run-time set-up. Similar to the Executionware architecture, it consists of static, non-cloud entities and components deployed in the cloud. The set of static components is comprised of the *Cloudify configuration*, the *user interface*, and *recipes*. Cloudify distinguishes

two flavours of recipes. One that describes a Cloudify application and another one that describes the individual services forming in an application.

Configuration All Cloudify configuration is file-based. That is, in order to use and re-use a configuration, a user has to have access to the same file system. The configuration files mainly contain the following information: *(i)* Which clouds are known to the system and what is their name. Cloudify supports multiple clouds from the same provider or from the same type as long as they are named differently. Names can be set freely by the Cloudify operator. *(ii)* Each cloud has its configuration in a single directory on disk. Within this directory reside all binaries and configuration files for that cloud.

User interface The user interface (UI) provides access to the Cloudify configuration and enables the bootstrapping of clouds and the deployment of applications (cf. *Cloudify Lifecycle* section below). The basic implementation of the user interface shipping with Cloudify is shell-based and hence, hard to share among multiple users. The user interface defines the lifecycle events for applications as discussed below.

Application recipes An application recipe defines application properties and the services, the application depends on. The recipe gives a name to the application and it allows the definition of environment variables shared by all of its services. Apart from that the application recipe defines interdependencies between the individual services. These are reflected when the application is deployed.

Service recipes The service recipe defines attributes of an individual service (all instances of a component). This includes the definition of the service name, the maximum and minimum possible instances running this service and actions for lifecycle events (cf. *Cloudify Lifecycle* section below).

Run-time Entities

In order to be able to make use of a cloud, i.e. deploy applications on a Cloudify cloud, it is necessary to *bootstrap* the cloud via the user interface. Cloudify uses a cloud driver to access the API of the cloud provider and uses it to allocate one or multiple management VMs. Afterwards, the Cloudify UI connects to the management VMs and installs the Cloudify controller components. The most important of these components with respect to their use in PaaSage are the *Elastic Service Manager (ESM)* and the REST service. Their interplay allows deploying applications and services as outlined below. On newly allocated non-management VMs, Cloudify installs the *Universal Service Manager (USM)*.

REST service The REST service is provided by any management VM. It offers a HTTP-based contact point for deploying new applications, deploying new services within an application context, scaling out/in existing services or shutting down entire services and applications.

Elastic Service Manager For any bootstrapped cloud, there is always one ESM running on one of the management VMs. This ESM receives deployment and scaling requests from the REST service of any management VM and maps them to invocations of the driver of the cloud it is running in. The ESM further provisions the necessary binaries of an application once the driver has allocated a virtual machine for running the application and once the USM has been installed.

Universal Service Manager The USM deals with the task of managing and monitoring service instances to be deployed at individual virtual machines. It defines the lifecycle for services and controls its execution. Moreover, it deploys monitoring probes as defined in a service recipe. Finally, it spreads the monitoring data so that it can be processed in the management VMs.

Monitoring

Cloudify comes with built-in monitoring capabilities on both per VM and per application basis. The request to monitor dedicated parameters is specified in a service recipe. Cloudify uses a JMX built-in plugin that collects JMX attributes over a JMX-RMI connection and exposes them as service monitors for each application.

By default, monitored data can be accessed through a Web interface provided by each management VM. This console provides a dashboard containing basic information for the VM status, such as deployment health and resource (CPU and memory) utilisation. In addition, it provides information about the status of each service in the deployment, the number of service instances, a list of machines and their services, as well as a list of application services and their corresponding health.

Finally, Cloudify comes with support for simple scaling rules that enable a Cloudify application to dynamically scale out (acquire more instances) and scale in (release instances) services when configurable conditions hold. For instance, when the number of requests to be processed by a service per interval reaches a certain limit.

Cloudify Lifecycles

Cloudify knows three interlinked lifecycles. Those for *applications*, those for *services*, and those for *service instances*. In addition, there are some lifecycle-

like features available for cloud bootstrapping. We describe each of these lifecycles in the following paragraphs.

Application lifecycle The application lifecycle is hard-coded into Cloudify and cannot be configured by users. The basic mechanism to trigger a state transition is to either type commands at the user interface or by accessing the REST service of a management VM. Applications go through the four phases *install* (waiting for the installation of an application), *Services installation* (bringing up all services of the application in the right order), *uninstall* (waiting for the application to be uninstalled), and *Services uninstallation* (bringing down all services of the application).

Service lifecylce The service lifecycle defines two events, namely *preService-StartEvent* called prior to starting the service instances and *preStopService* called after service instances have been shut down, but before for the virtual machines allocated by the services have been shut down.

Service instance lifecycle At each virtual machine allocated and operated by Cloudify, the deployment process installs the *Universal Service Manager (USM)* that controls the lifecycle of each service instance and moreover steers the probes that take care of the monitoring. Figure 5 lists almost all of the lifecycle events available for service instances and shows examples when they may be used. Note that the events of the service life-cycle (preStartService and preStopService) are executed prior to `init` and after `shutdown` of all service instances.

Cloud lifecycle The bootstrapping mechanism used for clouds enables the use of lifecycle-like functionality for clouds and management VMs. By default, Cloudify copies a `boostrap-management.sh` file to any management VM. While the name of the file is fix, its content may be adapted as needed. In addition, a `pre-bootstrap.sh` and a `post-bootstrap.sh` file may be specified. These are executed prior to and after `boostrap-management.sh`.

4.2 KairosDB

In this section, we present basic capabilities and features of KairosDB, a Java-based, open-source, time-series database implementation that the Executionware uses as a key component of the monitoring infrastructure. The decision for KairosDB was made for the following reasons: *(i)* In contrast to many other time-series databases, KairosDB is open-source and hence, allows extensions

	name	description/example
1	`init`	invoked when the USM starts; may be used for validating system environment;
2	`preInstall`	may be used to get service binaries, e.g. by downloading
3	`install`	may be used to unzip and install service binaries
4	`postInstall`	may be used to adapt configuration files according to environment
5	`preStart`	may be used for checking that required operating system files are available, like files, disk space, and port
6	`start`	launches the external process; mandatory event
7	`startDetection`	may be used to notify USM that a started event is ready for use
8	`stopDetection`	may be used to notify USM that the service instance has stopped, be it intended or unintended
10	`postStart`	may be used to register service instances with a load balancer
11	`preStop`	may be used to unregister service instance at the load balancer
12	`stop`	may be used to add manual stop logic
13	`postStop`	may be used to release external resources
14	`shutdown`	should be used to perform any required cleanup before the USM instance shuts down

Figure 5: Selected life-cycle events for service instances

and modifications. *(ii)* KairosDB's main open-source competitor, OpenTSDB[3], mainly targets the generation of metric graphs. Here, data is manipulated to make good-appearing and meaningful graphs. For example, interpolation is used to fill in holes left by missing data points. Apart from the fact that this is not in the primary focus of PaaSage, evaluations also show that KairosDB outmatches OpenTSDB by far: To this end, using the default row size in OpenTSDB's HBase we are able to store one hour of monitoring data, while using KairosDB in combination with Cassandra [10] we are able to store three weeks of data in a single row. *(ii)* KairosDB's main capabilities reflect critical requirements as defined in deliverable D1.6.1 [2]. In particular, these capabilities are:

1. KairosDB is a fast, distributed, and scalable time series database.
2. KairosDB in combination with Cassandra supports storing at millisecond granularity. This allows the capturing of $40,000$ points of data per second with just a single Cassandra node.
3. KairosDB provides support for both raw data as well as aggregations.
4. KairosDB supports grouping by tags, time range, or value to get filtered data.
5. KairosDB supports import/export of data in (compressed) JSON files.
6. KairosDB provides diagrams of the aggregated metrics.

The following sections introduce the main features of KairosDB that will be exploited by the Executionware prototype monitoring infrastructure.

General Architecture

KairosDB is a distributed database. That is, it can be configured such that multiple KairosDB instances running on different (physical or virtual) nodes cooperate and provide a common view on the data stored by all these nodes. KairosDB uses a configurable underlying data store that is used to persistently store the database content. Yet, KairosDB uses this datastore not only for achieving persistence, but also for spreading the data across all participating nodes.

In the default configuration, KairosDB uses Cassandra [10] as its data store. Cassandra in turn, realises a consistent hashing ring [9] to interconnect its instances and to spread data.

[3] `http://opentsdb.org`

aggr.	description
avg	average value
dev	standard deviation
div	each data point divided by a divisor
histogram	calculates a probability distribution and returns the specified percentile for the distribution
least_squares	two points for the range representing the best fit line through the set of points
max	largest value
min	smallest value
rate	rate of change between pair of data points
sum	sum of all values

Figure 6: List of KairosDB's built-in aggregators.

```
{
 "cache_time": 0,
 "start_absolute": 1391464800000,
 "end_absolute": 1391551200000,
 "metrics": [
  {
   "tags": {},
   "name": "kairosdb.jvm.total_memory",
   "aggregators": [
    {
     "name": "div",
     "divisor": "1024"
    },
    {
     "name": "avg",
     "align_sampling": true,
     "sampling": {
      "value": "1",
      "unit": "milliseconds"
     }
    }
   ]
  }
 ]
}
```

Figure 7: Example for aggregators on the `jvm.total_memory` metric. First the `div` aggregator divides data points of the specific time interval by 1024. Then, `avg` calculates the average value of the divided data point values.

Monitoring Support

KairosDB can store data points of any type of metric. KairosDB requires the following fields for the inserted data points: Metric name, value, timestamp, as well as other tags to distinguish between the different measurements. The latter may include for instance VM id, cloud provider, VM-type, application fields name, task name.

Aggregation

KairosDB comes with a series of built-in aggregators (cf. Figure 6) that perform an operation over all data points of a specific metric that exist in the sampling

period. All aggregators, except `rate` and `div` allow downsampling, i.e., reducing the sampling rate of the data points and aggregating these values over a longer period of time.

Aggregators can also be chained: The output of one aggregator is sent as input to the next. The order is retrieved from a JSON file that is also used to actually define the aggregators. For example, the JSON query file shown in Figure 7 includes two aggregators on the `jvm.total_memory` metric. The first divides data points of the specific time interval by 1024 and the second calculates the average value of the divided data point values. In addition to the chaining capabilities, aggregators allow an easy modification by the user yielding added value functionality.

4.3 Provisioning of 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.

GWDG's OpenStack Testbed

GWDG provides its in-house *GWDG Compute Cloud* with virtual machine images with various wide-spread Linux operating systems pre-configured for instant usage. Using these images, users can create virtual machines and perform operations on them including suspending, pausing, rebooting, and delete.

The platform further allows assigning public IP addresses to virtual machines and to define firewall rules. Finally, the *GWDG Compute Cloud* also provides basic monitoring functionalities that enable the retrieval of data such as CPU utilization, free memory, networks I/O on a per virtual machine basis.

The PaaSage consortium has access to the two GWDG OpenStack installations running OpenStack Essex and Folsom respectively. The Essex-based Cloud has 6 physical hosts of which 4 are available for hosting virtual machines and 2 are used for cloud infrastructure management. Each host has 64 physical cores, totaling 384 cores. The recent Grizzly-based Cloud has 8 physical hosts totaling 512 physical cores of which 320 are available for running virtual machines. Each of our physical host has 256 GB of RAM, which totals to 1536 GB for the Essex-based cloud and 2048 GB of RAM for the Grizzly-based Cloud. GWDG can extend its Cloud resources using hardware-on-demand basis, with 9 additional physical hosts each having 64 cores and 256 GB of RAM at its immediate disposal. All machines are operated in the data center of GWDG in accordance with current data protection and privacy regulations.

Flexiant Testbed (FLEX)

Flexiant built and made available a testbed in the in-house datacentre at Flexiant premises in Edinburgh. Having an own testbed for the consortium provides the best opportunity for PaaSage components to be integrated to the cloud orchestration layer, have access to platform data and to know what is happening at the physical level with RAM, CPU and storage. Further, this allows for an easy version upgrade when there are specific new FCO features developed that can be used for the benefit of PaaSage or to be shown in demos.

Flexiant's testbed consists of ten compute nodes with different hardware equipment. While each node has two network interface cards with 1 Gigabit Ethernet links, the memory equipment and number of cores varies: four nodes run a quad core CPU and 64 GB RAM, two nodes run a dual core CPU with 8 GB RAM, two nodes run a dual core CPU with 16 GB RAM, one node runs a octa core CPU with 64 GB RAM, and one node runs a single core CPU with 8 GB RAM. One network interface card of each node is linked to the SAN node, while the other connects to the FCO server running two quad core CPUs, two 1 TB Raid-1, and 16 GB of memory. Using this hardware platform, Flexiant is capable of running over 450 virtual machines with 8 GB of RAM allocated and further 230 virtual machines using 4 GB of RAM.

4.4 Conclusions

In this section, we have discussed the technical foundations of the Executionware implementation. In particular, we detailed the Cloudify framework that PaaSage uses as its primary deployment tool. We conclude that Cloudify has an architecture that resembles a lot the Executionware architecture which enables a quick realisation of support for basic use cases as envisaged in the M18 prototype. Nevertheless, Cloudify's single-cloud capabilities and its limited service orientation require extensions for its use in PaaSage's more sophisticated use cases. First and foremost the handling of PaaSage's complex scalability rules [4] requires a modification of Cloudify's built-in monitoring infrastructure (cf. Section 5). In particular, a suitable user interface has to be provided for that the Upperware can address cross-cloud service-level objective violations.

The use of KairosDB as a time-series database provides us with a distributed, highly scalable database perfectly suited for handling monitoring data. Its distributed nature enables that it spans multiple cloud platforms. For the M18 prototype, KairosDB was integrated into the Executionware and tightly coupled to the bootstrapping of clouds over the Executionware frontend. Moreover it constitutes the access point for the meta-database to retrieve monitoring data. The use of a distributed database spreading over multiple clouds is a first step towards the real multi-cloud capability of the Executionware (cf. Section 6).

5 Application Deployment and Cloud Management

One of the primary tasks of the Exectionware is to bring to execution, i.e. deploy the application description received from the Upperware sub-system. In order to deploy a new application, the Executionware requires information on the clouds the application shall be deployed on, basic information on the underlying execution platform, e.g. the virtual machine, consisting of operating system, type and size of virtual machines, memory requirements, etc. It may also require knowledge about available virtual machine images. Apparently, not all of these selection criteria are available on all types of cloud platforms. In particular, PaaS environments do not offer the concept of a virtual machine [8]. Nevertheless, the Executionware has to be able to deal with all of them.

Information such as the available clouds (in contrast to cloud instances) and the virtual machine images per cloud have to be accessible for the Upperware so that they can be used in the reasoning process and further be integrated in the deployment configuration passed from the Upperware to the Executionware. In order to enable a seamless development, testing, and first integration of the individual components, the M18 prototype of the Executionware provides a cloud registry where information about clouds and images can be stored. Section 5.1 presents a high-level view on this registry. Mainly, it outlines what information is available. For a detailed specification of how it is accessed, we refer to the documentation of the code.

For the deployment in the M18 prototype is based on Cloudify (cf. Section 4.1) the data passed over the Upperware-Executionware API eventually has to be transformed in one or multiple working Cloudify configurations. For the concrete API to be used is dependent on deliverable D2.1.2 [4] and further, the integration of the various PaaSage components is scheduled after M18, Section 5.2 mainly discusses a Cloudify-like API for the Executionware and further provides an outlook on a CAMEL-based interface. Section 5.3 concludes the section with a summary of the current implementation status of the Executionware frontend.

5.1 The Executionware Cloud Registery

As stated in the terminology section of this document, a cloud is the URI of a cloud instance combined with the necessary credentials to access the cloud instance. The frontend of the PaaSage Executionware comes with a service that allows the registration of clouds as well as their internal properties such as available virtual machine images and selectable hardware configurations. The service is backed by a graphical, Web-based user interface that can be used to register

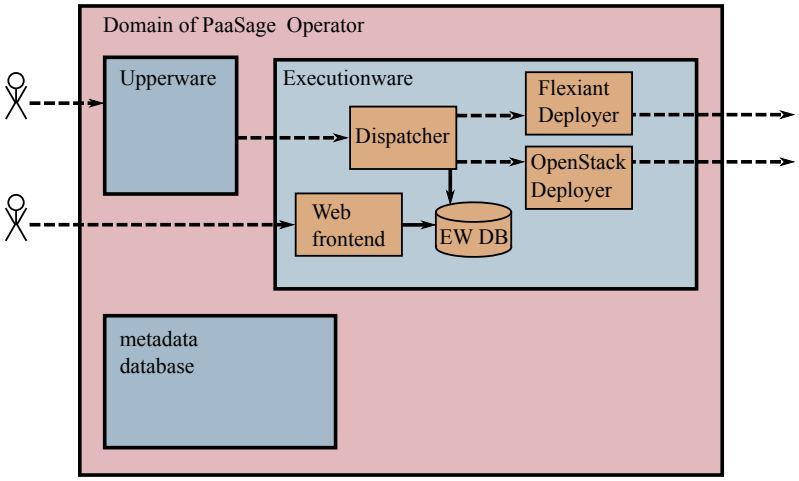

Figure 8: Overview on the basic approaches to access the Executionware (in particular, the dispatcher) through a HTTP interface and through a Web-based GUI. The GUI can be used to register new clouds; information on the available operating system images and the virtual machines and so on may be added on a per-cloud bases. The deployment (cf. Section 5.2) is exclusively dealt with over the HTTP-based interface and does not come with a GUI.

new clouds and properties of a cloud. In addition, the interface can also be queried by the Upperware over a HTTP interface. Figure 8 sketches the static part of the Executionware as shown in Section 2 in combination with the registry. In the following, we present the capabilities of the registry by first dealing with the management of clouds and then presenting the way additional mechanisms are treated.

Managing Clouds

The interface allows the following four operations to take place for a cloud:

1. `register` Provides the necessary information to access the cloud. This includes the definition of the cloud platform (currently Flexiant and OpenStack) and further parameters that may depend on the type (such as URI, username, password, keypair, ...). At success, the operation returns a unique identifier for that cloud. Figure 1 shows some of the configuration properties associated with a cloud.

2. `start` Starts a cloud as it was defined by `register`. This includes the startup of a Cloudify management VM and co-located on that virtual

property	description
name	the name as seen by the user
driver	the driver to use for accessing this cloud; dependent on the cloud platform to be supported;
driver configuration	configuration parameters for the driver such as URI of the cloud instance;
#management VMs	desired number of management VMs

Table 1: Some of the configuration properties associated with a cloud

machine a KairosDB instance (cf. Section 6). Only after this step applications may be deployed to a cloud. The operation results in an error when the cloud is already started or is currently in the progress of being stopped. Otherwise, the URIs of the management VMs are returned.

3. `stop` Stops the cloud. This includes the stopping of all applications (component instances) currently deployed on that cloud. It also removes the management VM(s) and the TSBD instance(s). This operation returns an error when it is not possible to stop the cloud or if it has not yet been started.

4. `delete` Removes the cloud from the database. This operation can only be executed when the cloud is not started (i.e. stopped).

`register` and `delete` are purely local operations that do not interact with any cloud provider and hence do not require Internet access. Only `start` and `stop` have impact on the cloud platforms and may cause costs. Please note that for the M18 prototype, we have not foreseen any update functionality.

Registering Cloud-specific Data

The Web interface enables the registration of additional data with a particular cloud. This particularly includes hardware configurations and operating system images. For each image, the operating system, its version, and the system architecture have to be specified. An image may further be tagged with keywords to identify the software bundles already installed on the image. Regarding hardware configuration, it is possible to define 3-tuples on a per cloud-basis. Such a tuple consists of #cores, the amount of RAM capacity, and the storage capacity to be used with a virtual machine.

When deploying an application, the user has to select the cloud, the virtual machine image to use for that application, and the hardware configuration for the virtual machine.

5.2 Deployment of Applications

Basically, the deployment of an application is triggered by the Upperware. In order to execute the deployment, the Upperware invokes the HTTP API of the Executionware frontend installation. Doing so, the Upperware passes a data structure in JSON format that describes the application, its artefacts, and the rules necessary for monitoring. In addition, the data may contain information on which sensors to bring out on a per-component basis.

In order to enable a step-by-step development of the Executionware software and furthermore, enable a step-wise integration of Executionware and Upperware functionality, the primary interface of the M18 prototype of the Executionware frontend is a HTTP-based wrapper around the Cloudify deployment functionality. The definition of a CAMEL-based interface between Upperware and Executionware has been started, but its finalisation is beyond M18. In particular, it is dependent on D2.1.2 [4]. In the following, we briefly discuss both APIs with the Cloudify API being usable and the CAMEL-based API being in early draft state.

Cloudify API

For deploying Cloudify recipes for both services and applications we face the same problem as for the cloud management: Cloudify's file-oriented interface has to be transferred to something that is much more service-like and hence can be used similar to the cloud API presented in Section 5.1.

For that reason, in addition to the cloud-specific data already discussed, the Web interface of the Executionware frontend also enables the registration of archive files, e.g. zip files, that contain a Cloudify-specific application recipe. In order to enable the re-use of certain applications components such as a Servlet container (e.g. Tomcat) or a database, we further allow the registration of individual services recipes. Again, these can be registered via archive files. Then, application recipes can reference those services.

The registration of both service and application files happens via the Web-based user interface. In addition, the operations that work with these files, in particular the deployment of applications, can also be triggered over the Web interface.

In order to enable the Upperware to access the very same functionality, a further HTTP-based interface is required. The operations that have to be offered

HTTP op	URL	description
POST	`./`	register new application/service; requires that archive file be passed; returns an identifier that is unique for this Executionware instance
GET, PUT, DELETE	`./$Id`	get/update/delete registered application/service, updating may be used to re-configure the entity updating may also be used for deployment

Figure 9: HTTP-based API for registration of Cloudify services and applications as well as deployment and scaling of applications. The root directory `./` is `/cf/applications/` for applications or `/cf/services` for services.

to the Upperware include starting, stopping, and scaling of application. The interface URIs are listed in Figure 9. A detailed specification of the data and data format expected for each operation as well as the specification of archive files containing recipes is subject to ongoing work.

CAMEL-based Interface

Even though the definition of the final Upperware-Executionware API has only been targeted for the integration period (M18–M21), the work has already been started. The basic information required by the Executionware for being able to deploy an application comprises *(i)* a definition of the software artefacts being used and how they are assembled to software components. *(ii)* Information and logic of how to install and configure a software artefact. *(iii)* The specification of what components shall be deployed on what cloud and how many instances shall be used of a particular component. In addition, *(iv)* information on what hardware and operating system the component shall be executed, and *(v)* how component instances shall be grouped.

Then, the Executionware has to map the data received to the clouds, virtual machine images, and hardware configurations available. For instance, a component that requires *Linux* may be matched to any Linux image while those requiring *Ubuntu* may be matched to any images with any Ubuntu version. Finally, the Executionware will create valid Cloudify configurations and deploy them afterwards. In order to control the steps executed by the Executionware, they will be reflected in the Cloudify API presented in Figure 9. A first CAMEL-enabled version of the Executionware frontend including data and data format is sketched in Annex A.4.

5.3 Conclusions and Status of Implementation

In this section, we have recaptured the tasks and the requirements of the Executionware frontend whose main task is to serve as an access point for the Upperware in order to deploy new applications or to change the deployment of existing applications. For the M18 prototype, we have implemented a custom, Cloudify-compatible driver for the Flexiant Cloud Orchestrator. The other targeted cloud platform, OpenStack, is supported out-of-the-box by the `jclouds` middleware suite[4] that ships with Cloudify.

In addition, we implemented a Web-based front-end in order to support the configuration of clouds and to enable Upperware access to the Executionware functionality. So far, a Cloudify-compatible API is implemented, while the specification of a CAMEL-compatible API is still subject to ongoing work.

Concluding, the implementation status of the Executionware frontend at M18 matches the goals sketched for the infrastructure entities in Section 3.1, namely

1. to receive deployment information using a HTTP-based interface
2. to process that deployment information and deploy applications
3. to enable deployment to a **single** cloud that either runs Flexiant Cloud Orchestrator or OpenStack as cloud platform.
4. to derive the right virtual machine image and virtual machine configuration to use from the data passed by the Upperware

Finally, the fact that Cloudify deploys a Universal Service Manager on each acquired virtual machine, also covers the goal we defined for runtime entities, namely to provide an entity that functions as a communication endpoint for the infrastructure entities. Goals defined for monitoring functionality are related to the time-series database and hence related to Section 6.

6 Application Monitoring

Beside deployment of applications, monitoring the execution of each individual component instance and virtual machine is a core feature of the Executionware. Here, the monitoring modules of the Executionware provides a two-fold functionality. On the one hand, it has to process monitoring data on the fly so that a quick reaction to violations becomes possible. On the other hand, it has to feed enriched and accumulated data back to the meta-data database for a later off-line evaluation.

[4] `http://jclouds.apache.org/`

The use of a time-series database is useful and appropriate as particularly time-stamped, real-time data streams as emitted by monitoring sensors are not well-suited for the meta-data database. This is due to the following reasons: *(i)* Relational databases are not efficient at handling time-stamped data [7]. *(ii)* The sheer amount of data puts an unnecessary load on the meta-data database, as *(iii)* the system is in almost all cases interested in obtaining statistical measures over raw data and not on the raw data themselves [2, 6].

The primary focus of the M18 prototype is on two features: *(i)* the provisioning of monitoring data to the time-series database. That is, probes have to be installed at the acquired virtual machines and the monitoring data has to be targeted towards the time-series database. *(ii)* The data arriving in the time-series database shall be accumulated. *(iii)* In addition, the time-series database has to collect and consolidate the monitoring information available in multiple clouds. *(iv)* It provides a mechanism to feed collected, filtered, and accumulated data back into the meta-data database.

The reasons for the use of KairosDB as a technical foundation for the time-series database have already been clarified in Section 4.2. The distributed capabilities of KairosDB enable an easy consolidation of monitoring data stemming from different clouds. In Section 6.1 we discuss the integration of the time-series database with the Executionware frontend and in Section 6.2 we consider its interactions with the meta-data database. Section 6.3 briefly summarises the implementation status for the M18 prototype.

6.1 Integration to With Executionware

The integration of KairosDB with the Executionware happens at several levels. First of all, instances of KairosDB have to be installed, deployed, and configured whenever a cloud is bootstrapped by the Executionware frontend. Then, monitoring data retrieved from the Cloudify sensors in a cloud have to be fed into the time-series database. Finally, it has to be selected which data shall be aggregated by the time-series database.

Provisioning and Bootstrapping

For the data collection part of the monitoring infrastructure has to be available already when an application is deployed, it is only reasonable to deploy at least one instance of KairosDB when bootstrapping a cloud via Cloudify. In order to achieve a global view on the collected data, the TSDBs running in multiple clouds, but owned by the same tenant have to be interconnected.

The PaaSage Executionware exploits Cloudify and other Executionware mechanisms for the deployment of KairosDB instances each time a cloud is started

using the Executionware frontend. Consequently, it also undeploys this instance whenever a cloud is shutdown.

On a technical level, the Executionware provides an own application recipe for the time-series database consisting of an application description *TSBD* that depends on KairosDB and Cassandra as services. In addition, we provide recipes for each of the two services. This enables us to use the `bootstrap-post.sh` hook (cf. Section 4.1) for deploying a KairosDB instance with each management VM.

Feeding Data to the Database

For the M18 prototype, PaaSage's Executionware exploits the monitoring capabilities provided by Cloudify (cf. Section 4.1). In particular, we use Cloudify's sensors to retrieve necessary data from the virtual machines and component instances. In addition, each KairosDB instance is accompanied by a daemon process whose task is to periodically collect data from the Cloudify sensors and push that data into KairosDB.

In order to simplify the data transfer, we provide an own interface to KairosDB that extends the basic interface and eases pushing data to and pulling data from the database. The interface may be accessed either via telnet or via REST. Regarding the first interface, the syntax is concise and easy to use:

```
put <metric name> <timestamp> <value> <tag>+ /n
```

The REST API can be exploited for adding measurement data by sending JSON-formatted data through HTTP. For adding large amounts of data it is possible to send `gzip`ped JSON files and upload them with the HTTP content type set to `application/gzip`. The general structure of the JSON file is shown in Figure 10. The first entry inserts three data points of one metric, while the second adds a single data point to another metric.

Using this approach the Executionware can push both management and application specific information into KairosDB. In addition to using the default sensors shipping with Cloudify our approach is further open for defining custom sensors which can then be integrated them with the Executionware and Cloudify mechanisms. As a proof of concept, we implemented a jBoss[5] monitoring plugin that functions as a sensor for PaaSage. It supports the following metrics:

busy_t: threads currently busy with processing HTTP requests

http_t: current total number of threads resolving HTTP requests

total_t: total number of threads used by the server

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

```
[
    {
        "name" : "CPU-usage",
        "datapoints" : [
                    [1391464800, 87%],
                    [1391464920, 73%],
                    [1391465040, 92%]
            ],
        "tags" : {
                "Cloud_provider":"Flexiant",
                "VM-ID" : "1234",
                "Application_componentID":"2345"
        }
    },

    {
        "name" : "Memory-usage",
        "timestamp" : 1391464800,
        "value" : 469,
        "tags": {
                "Cloud_provider":"OpenStack",
                "VM-ID" : "1235",
                "Application_componentID":"2346"
        }
   }
]
```

Figure 10: General JSON structure for pushing data to KairosDB over the REST interface. The first entry inserts three data points of a metric denoting CPU usage, while the second entry adds a single data point to a memory utilisation metric.

avail_m: free memory available for the JVM running the server

Data Replication

From the deployment scenario follows that there is at least one instance of KairosDB running on each bootstrapped cloud. All virtual machines from one cloud report their monitoring data to a TSDB running in the same cloud as they do. In order to be able to aggregate data issued from different clouds, it is necessary to interconnect the different TSDB instances. In fact, such an interconnection functionality is provided by the Cassandra platform underpinning KairosDB.

Basically, instances of Cassandra form a consistent hashing ring [9]. Figure 11 depicts such a set-up with three KairosDB instances running in three different cloud environments namely Flexiant, OpenStack, and Amazon EC2. In order to be equipped against failure, disconnection or user-triggered undeployment of TSDB instances it is advisable to apply data replication.

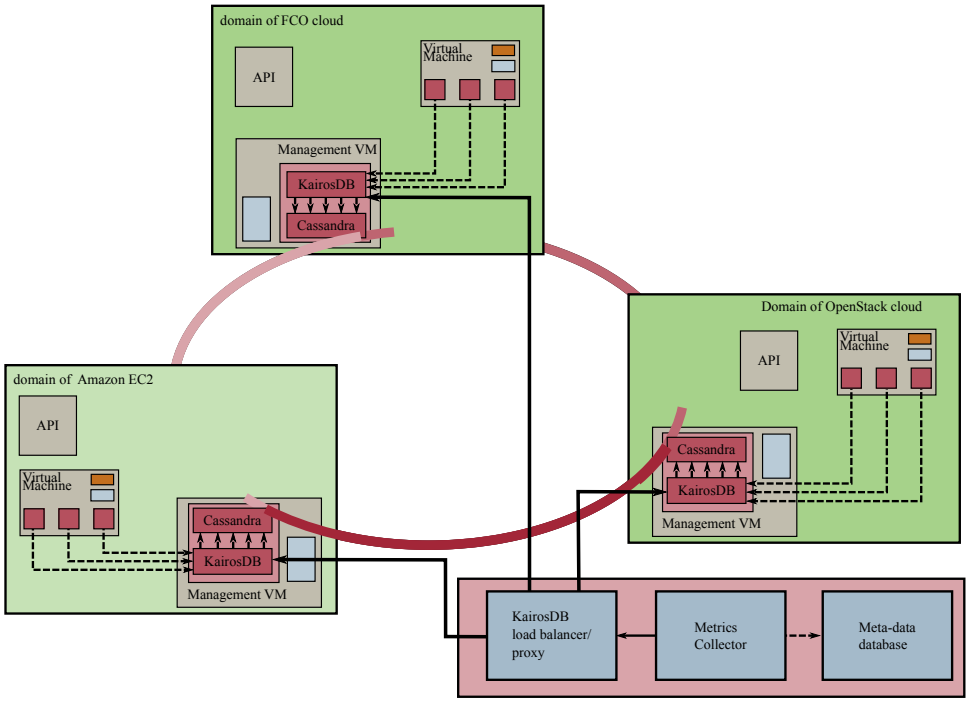

Figure 11: KairosDB multi-cloud replication group with load balancer

When writing to Cassandra, the write must succeed in a set of replicas before returning an acknowledgement to the client application. The size of the replica set is configurable and mainly determines the consistency level of the data store. Majority quorum is a common scenario ensuring strong consistency, yet still tolerating some level of failure. A majority quorum is calculated as rounded: $(\#instances/2) + 1$

In the example of Figure 11 three instances exist so that a majority quorum requires 2 successful reads/writes in order to tolerate the failure of one replica: If a write occurs in the Flexiant VM either the Amazon or OpenStack KairosDB instance has to acknowledge it before the operation is considered successful.

6.2 Integration with Meta-data Database

In order to durably persist important information and make it available to other sub-systems of the PaaSage platform, the data contained in the TSDB instances and shared among them finally has to be stored in the meta-data database. For that purpose, we introduce the *Metrics Collector* as an additional software entity. The Metric Collector runs in the domain of the PaaSage operator and periodically queries the REST API through a load balancer of the KairosDB instances.

For KairosDB instances are replicated, the Metrics Collector can retrieve the required information for the whole application (spanning multiple clouds) when querying a single TSDB instance. In order to avoid overloading a single instance we add an HTTP load balancer responsible for balancing the HTTP read requests according to a configurable policy (cf. Figure 11).

The data retrieved by the Metrics Collector depends on the Service Level Objectives for the specific application and its components. Such objectives map to particular composite/aggregated metrics and thus can be used in defining which aggregators shall be used in the time-series database instances. The tags of the data points facilitate the retrieval of specific information such as the execution time of a specific deployment of a multi-cloud application, the CPU utilisation of a particular virtual machine, the availability of application components in a time interval, etc.

6.3 Conclusion and Implementation Status

In this section, we have recaptured the motivation, the tasks, and the requirements for the use of a time-series database. Its main task is to collect required monitoring data from all initialised component instances, to consolidate this data over multiple clouds, and to finally feed back the data to the meta-data database. For the M18 prototype, we have integrated the Java-based open-source KairosDB time-series database in the Executionware environment. The Executionware frontend installs an instance of KairosDB together with an instance of the Cassandra distributed storage on all of Cloudify's management VMs. For the Cassandra storage instances form a consistent hashing ring, the full data set is available from all KairosDB instances. We further implemented the necessary logic to feed Cloudify's monitoring data to KairosDB and in consequence to the Cassandra storages. We also realised the Metrics Collector component that is responsible for pulling the data out of KairosDB and storing it in the meta-data database. Our implementation of a jBoss sensor shows the extendibility of our monitoring approach.

While both the Metrics Collector and the KairosDB aggregators are capable of doing their respective work, the automisation of their configuration is still subject to ongoing work. In particular, deriving the necessary monitoring sensors from the scalability rules attached to application specification (cf. deliverable D2.1.2 [4]) as well as deriving required aggregators from service level objectives is an important task for the upcoming period.

Concluding, the implementation status of the monitoring prototype at M18 matches the goals defined for the monitoring entities in Section 3.1, namely

1. to install specified sensors.

2. to retrieve values from dedicated sensors.

3. to collect monitoring data on a per-cloud basis.

4. to provide filtered and accumulated monitoring data to the meta-data database.

Even though not mentioned in this section, also the demand to support scale-out on a per-cloud, per component instance level is realised with the current prototype. This is due to the fact that this feature is built-in to Cloudify for simple applications and simple scalability rules.

7 Workflow Applications

Large-scale scientific workflows, such as computational fluid dynamic (CFD) and Molecular Docking (MD), are compute-intensive applications that consist of many tasks with interdependencies. The requirements of workflow execution engines make the use of cloud platforms attractive. In addition, the various optimisation parameters that exist in such a highly complex application values the use of PaaSage.

In the following, Section 7.1 first discusses the case for workflow execution in PaaSage and identifies a generic approach of how to integrate it with the other parts of PaaSage and in particular PaaSage's Executionware. The workflows considered in PaaSage and hence in this document exclusively deal with AGH's `HyperFlow` workflow engine [1] that we introduce in Section 7.2. Finally, we present the status of the current implementation efforts in Section 7.3. More details about HyperFlow and the Montage application is available online[6] and in deliverable D2.1.1 [4]. References to the installation guidelines implementation status are available in the Annex.

7.1 The Case for Workflow Execution in PaaSage

The requirements of workflow execution engines make the use of cloud platforms attractive for various reasons: *(i)* The resource requirements of a workflow application strongly depend on the stage it is in. While computing stages require many compute nodes, a preparation or evaluation phase may require less. *(ii)* Depending on the stage of a workflow, the type of nodes required may differ. Currently, this mainly includes the amount of memory or storage capacity. In future cloud offerings, this may also target other hardware equipment such as GPUs available in the virtual machines. *(iii)* The optimisation goals for a

[6] `https://github.com/dice-cyfronet/hyperflow`

workflow may be different, again depending on its phase. For long lasting computations it is possibly not too important whether they take one week or one week and a day. In contrast, the rendering of the results should be reactive, at least generate smooth graphics, and allow user interactions.

All of these requirements make PaaSage attractive for workflow engines. Mainly because PaaSage can help to provide different optimisations depending on the workflow phase, and hence, enables the best possible execution of the workflow. In order to unleash the power of PaaSage, two demands have to be satisfied. First, a workflow execution engine has to be made cloud-enabled. Second, PaaSage has to be made workflow-enabled. The purpose of this section is to analyse how this can be done.

The cloud-enabling of applications is a rather technical task that includes the provisioning of installation and configuration scripts that can be executed automatically by the cloud deployer. The integration into PaaSage, however, has to happen both on a conceptual and on a technical level: As stated above, the requirements and specification of a workflow application may change depending on the stage of the workflow. In particular, this means that the PaaSage optimisation and deployment process shall be triggered whenever the workflow enters a new stage. At the current stage of investigation, we think that this is best realised when the workflow execution engine emits events using the Executionware's monitoring capabilties. Such a `new_stage` event will eventually reach the Upperware that will envisage a new deployment of the application.

7.2 The HyperFlow Platform

HyperFlow defines a model of computation and provides an enactment engine for complex distributed workflows. In HyperFlow, *workflow enactment*, i.e. the coordination of control and data flow between application components, is strictly independent from *execution*, i.e. the actual invocation of the application components. This is illustrated on the conceptual HyperFlow architecture in Figure 12.

A workflow in HyperFlow is defined in an abstract way—as a set of *processes* performing well-defined *functions* and exchanging *signals*. It is important to note that processes, functions and signals are *abstractions* of the workflow model, and should not be confused e.g. with physical OS processes. The HyperFlow workflow description is independent of the runtime environment and specific interfaces or instances of the application components.

The role of HyperFlow is to *(i)* process the workflow description provided as JSON files, and further to *(ii)* execute the ready tasks. The instance executing tasks is referred to as the `Executor`. HyperFlow is designed to plug-in

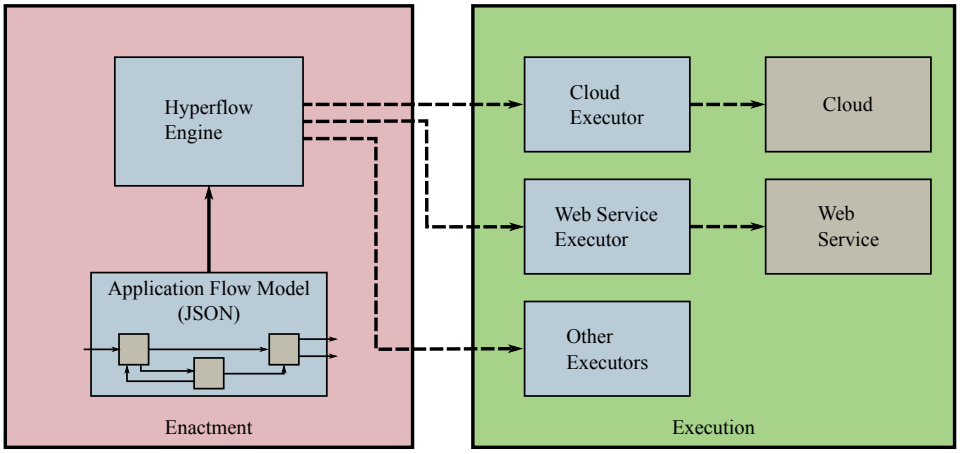

Figure 12: Conceptual architecture of the HyperFlow workflow engine.

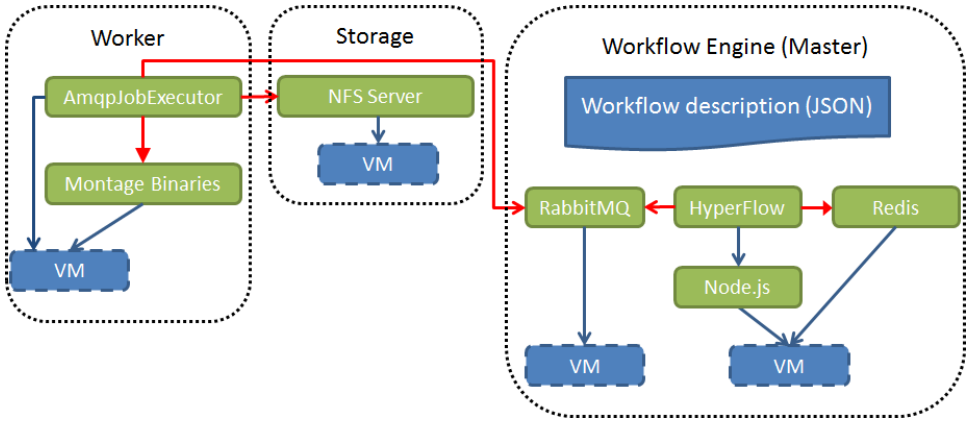

Figure 13: CPIM diagram [4] showing an example deployment of the HyperFlow execution components and the Montage workflow across multiple virtual machines.

various implementations of Executors. Besides cloud executor and Web service executor as shown in Figure 12, other available executors are the simple `Command` executor that executes the task locally as a command-line program, and `AMQPExecutor` that uses an AMQP broker to queue tasks from where `ExecutorWorkers` retrieve and then process them.

Figure 13 presents a CPIM diagram for the Montage application (cf. deliverable D2.1.1 [4]). In real-world settings this application can be easily composed of a pipeline of 10,000 tasks. The HyperFlow set-up shown in the figure uses the cloud executor to run the workflow. There, the HyperFlow engine shall be deployed on one virtual machine alongside its runtime compon-

ents `node.js` and `redis`. The cloud executor is composed of RabbitMQ, an AMQP compatible message queue deployed on a separate VM, and local workers (`AMQPJobExecutor`). Each virtual machine allocated for workers has to contain the worker binaries as well as the application binaries; that is in this example the Montage binaries.

HyperFlow enacts the workflow according to its description and, when the dependencies of a given task are fulfilled and it becomes ready for execution the HyperFlow engine generates a task specification, sends it to the RabbitMQ queue, awaits on the same queue for the completion of the task. Local executors on the worker nodes fetch tasks from the queue, execute appropriate Montage components, and send a message back to the queue to notify about the of a task.

7.3 Conclusions and Implementation Status

In this section, we have discussed the support for HyperFlow-based workflows in PaaSage's Executionware. We argued that workflows may particularly benefit from being used in PaaSage, because of their additional dynamic with respect to requirements. We also discussed a general approach to support workflows in PaaSage by first making the workflow execution engine cloud enabled and later on exploiting PaaSage's monitoring capabilities in order to integrate the two systems. For PaaSage makes use of the HyperFlow workflow engine, we further introduced the architecture of this software framework.

So far, the implementation with respect to workflow support has targeted the goal of making HyperFlow cloud enabled. The integration with the monitoring functionality is beyod the M18 prototype and hence, beyond the scope of this document.

The general concept for workflow execution in PaaSage is essentially to deploy the HyperFlow engine and the AMQP cloud executor as additional application artefacts driving the execution of the application itself. In order to facilitate deployment of HyperFlow modules onto virtual machines running on the cloud, we have prepared a cookbook for `Chef` infrastructure automation system. The cookbook contains `recipes` (scripts) for installation of the software modules.

The recipes are designed in such a way that they can deploy and configure all the required software modules together with all dependencies, by starting from a fresh operating system installation. The current version of the cookbook requires `Ubuntu 13.10` in the 64-bit version. Such an image is available out of the box on Amazon EC2 and on Flexiant cloud. Yet, the cloud-enabled version of HyperFlow is not *per se* bound to Ubuntu. By using Chef, it is straight forward to update the recipes to be compatible with other Linux distributions, such as Debian or RedHat, if such need arises.

The cookbook for HyperFlow modules and applications includes recipes for the HyperFlow engine and for the AMQP executor. In addition, the cookbook also contains recipes for the installation of example workflow applications. These include the Montage workflow that has already been presented in D2.1.2 [4] and the `MolecularDocking` application developed by USTUTT. More details on recipes are available in Appendix C.

8 Data Farming Applications

Data farming is a methodology based on the idea that by repeatedly running a simulation model on a vast parameter space, enough output data can be gathered to provide a meaningful insight into relations of model properties and behaviour. After it has been designed, a data farming experiment goes through the two phases simulation execution and progress monitoring as well as statistical analysis of the results. Similar to the workflow engine (cf. Section 7), each of these phases has dedicated requirements and can hence profit from PaaSage's sophisticated optimisation. Besides these workflow characteristics, data farming offers more capabilities for optimisation that we list in Section 8.1.

Afterwards, Section 8.2 presents an overview on the Scalarm[7] data farming platform provided by AGH. The Scalarm platform focuses on conducting experiments that follow this data farming methodology. It provides a complete platform for conducting such experiments with heterogeneous computational infrastructures. Its various features and already integratd cloud support make it particularly suited for an integration and exploitation in PaaSage. Finally, Section 8.3 sketches the approach of an integration and presents the implementation status of this undertaking.

8.1 The Case for Data Farming in PaaSage

Beside the opportunities already provided by the workflow discussion in Section 7, data farming offers even more chances to exploit the optimisation mechanisms provided by PaaSage.

A particular goal of the use of PaaSage for data farming experiments is on the one hand to automatise deployment and size of the various experiments. In addition, PaaSage can help in maximising hardware utilisation. For instance, it may decide whether it is better according to a specified metric to run multiple experiments parallel on a multi-core CPU or chose to acquire multiple single-core machines. Finally, supporting cloud environments to conduct data farming

[7] `http://www.scalarm.com/`

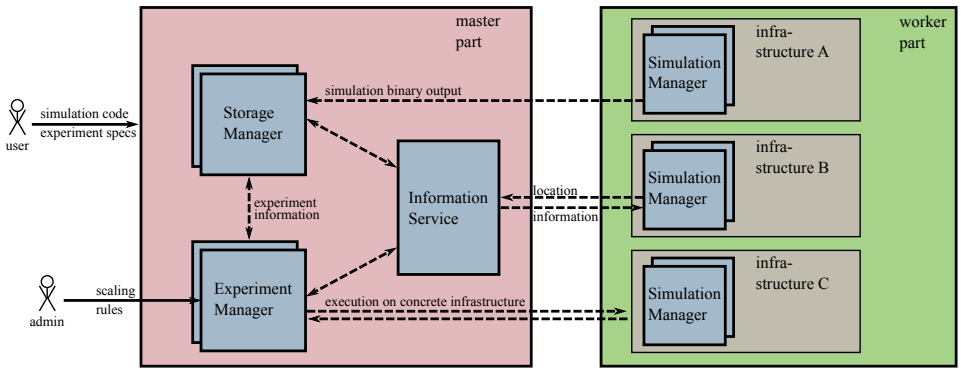

Figure 14: The Scalarm architecture.

experiments is especially important due to reducing the necessary upfront investment in hardware and provided elasticity.

Concluding, integrating data farming with PaaSage is an important step towards building a fully autonomous platform for data farming. Data farming may particularly benefit from the conceptually unlimited pool of resources provided by multiple clouds and the fact that PaaSage uses integrated scaling management.

8.2 The Scalarm Platform

The Scalarm platform supports all phases of a data farming experiment, from design through simulation execution and progress monitoring, to statistical analysis of results. It is suited for different sizes of experiments ranging from dozens to millions of simulations while maintaining computation efficiency through massive scalability. Finally, Scalarm is able to exploit heterogeneous computational infrastructure. It also supports private and public clouds as well as legacy infrastructures such as Grids.

Scalarm provides two distinguishing features compared to other data farming frameworks: First, it extends the usual batch-like experiment execution routine—the user submits an experiment as a single package, waits for all simulations to compute, and then analyses the obtained results—by the capability to expand the parameter space of currently running experiments. The decision to expand can be based on an on-line analysis of already computed simulations. Second, Scalarm enables the adjustment of computational resources dedicate to run experiments. Hence, the user may specify only a small parameter space with small amount of resources at first, and expand it later on.

The Scalarm architecture is depicted in Figure 14. Scalarm decomposes the functionality of supporting data farming into a set of loosely coupled services

responsible for managing. In particular, we can distinguish between the master and the worker part of the system. The *master* part consists of the Experiment Manager, Storage Manager, and Information Service. These can be located on separate resources, e.g. on different virtual machines, but not necessarily have to. The suggested resource configuration depends on the size of the conducted data farming experiment. The *worker* part of Scalarm solely consists of the Simulation Manager service.

Experiment Manager The Experiment Manager provides a GUI that processes all interaction between users and the Scalarm platform. As such, it provides a coherent view on the status and data of all running and completed data farming experiments. It enables users to prepare, schedule, and monitor experiments or to conduct statistical analysis on results. It is also responsible for dispatching simulations to Simulation Managers in the worker part.

Storage Manager Other Scalarm services, mainly Experiment and Simulation Managers use the Storage Manager as data back-end. The data stored there includes structural information about executed simulations and experiments as well as actual results of simulations. The Storage Manager comes with a built-in load balancer that hides the distributed nature of the service.

Information Service The Information Service represents a *well-known* entity that can be queried by all services to retrieve the location of any other service. Therefore during a service start-up, only the location of an Information Service instance has to be known.

Worker Service The worker service is an intelligent wrapper around the actual simulation runs. It can be deployed on various computational infrastructures, in particular on different cloud platforms. The Simulation Manager is designed to operate in highly dynamic and unreliable environments and can tolerate the failure of Experiment Manager and Storage Managers instances as well as network connectivity issues.

After having been scheduled to an execution environment such as a virtual machine, the Simulation Manager performs a four step execution loop. Once it finishes a loop it starts over with the next one. Starting one simulation after another reduces the resource acquisition time, e.g. virtual machine startup, and increases the overall experiment performance. The four steps are as follows: *(i)* The Simulation Manager retrieves a configuration and simulation binaries from the Experiment Manager. *(ii)* The Simulation Manager obtains a simulation configuration, i.e. a vector of input parameter values, which is a single element of

the experiment parameter space. *(iii)* The Simulation Manager executes the simulation possibly accompanied by a progress monitor. For performance reasons, the Simulation Manager may execute one simulation with multiple input vectors in parallel provided that sufficient CPU cores are available and machine load allows for it. *(iv)* The Simulation Manager captures the output and uploads it to the Experiment Manager.

8.3 Conclusions and Status of Implementation

In this section, we have discussed the case for data farming in PaaSage. We have seen that the workflow-like stages each data farming experiment passes through open the same optimisation potential as for workflows (cf. Section 7). At the same time, an advanced and modern data farming framework such as Scalarm allows optimisation capabilities beyond these of mere workflows.

In order to bring Scalarm to cloud platforms, the Ruby-based implementation of Simulation Manager was dropped and replaced by a Perl-based reimplementation. This is mainly, because, Ruby interpreters is not usually installed by default on many systems. This choice was approved by the fact that this Simulation Manager was able to run out of the box on virtual machines provided by Amazon EC2 and the cloud testbeds within PaaSage namely OpenStack and Flexiant.

From the PaaSage point of view, Scalarm is a single application consisting of multiple components. Each service has different requirements with respect to dependencies and scaling rules. The master part of the architecture coordinates the process of a data farming experiment, while the worker part, consisting of multiple instances of the Simulation Manager entity, is responsible for executing the actual simulation. To enable an integration with PaaSage, initial descriptions of each Scalarm service were prepared with the CloudML language. Due to that it is possible to allow an automatic deployment of the whole Scalarm platform on a static set of resources.

During an experiment, the Experiment Manager schedules Simulation Manager instances onto available computational resources. Furthermore, the Simulation Manager is responsible for executing user simulation on a particular computational resource, e.g. a virtual machine. As both entities share capabilities and tasks of Executionware Dispatcher and Execution Engine respectively, they are the primary target for an integration of Scalarm into PaaSage's Executionware. This is the primary goal of current development efforts.

In order to take the full advantage of functionalities offered by PaaSage, Scalarm has to be integrated at service description, scalability, and API level. The realisation sketch for such an integration is subject to Section 9.

9 Conclusion and Future Work

In this document we have presented fundamentals of PaaSage's Executioware and the current status of the Executionware implementation that is released as a prototype at M18. In the PaaSage architecture, the Executionware's main task is to bring to execution, i.e. deploy, applications that have been modelled by the application developer and optimised according to other constraints such as policies by the Upperware. In addition, it is the Exectionware that gathers necessary monitoring information, accumulates them, and finally feeds them back to the meta-data database so that the Upperware can make use of it. The prototype implementation is based on the open-source tools Cloudify and KairosDB.

Architecture-wise, the Executionware consists of three main parts: infrastructure, run-time, monitoring. Entities belonging of the *infrastructure* part, are hosted on the premises of the PaaSage operator and ensure that application deployment can successfully take place. They are implemented by the Executionware frontend module (cf. Section 5). Entities belonging of the *run-time* system run on the virtual machines deployed in the various clouds. Their main purpose is to enable a remote steering of the respective virtual machine, to intercept direct interactions with the cloud provider and finally, to control network traffic.

Finally, the *monitoring* entities of the Executionware are concerned with gathering monitoring data from the various virtual machines and component instances. This is done through sensors deployed with the virtual machines as well as monitoring information provided by the cloud operator. In addition, the monitoring part is concerned with collecting all the information, aggregating the data and eventually, feeding that data back to the meta-data database. The prototype collects monitoring data in the time-series database and uses the metrics collector component to feed it back to the meta-data database.

In addition to the basic functionality, the Executionware implementation is extended with dedicated solutions for workflow-based applications as well as data farming applications. In particular, initial work has been done to prepare the HyperFlow workflow engine and the Scalarm data farming platform for an integration with the Executionware.

9.1 Summary of Implementation Status

Table 15 lists planned features of the Executionware and also shows those features that have already been implemented. In particular, these features cover the scope targeted in Section 3.1. The combination of all these features enables the use of the PaaSage Executionware in the following manner:

1. Register supported clouds and images

	features of infrastructure entities	**status**
#1	single cloud deployment	done: M18
#2	cloudify-based API	done: M18
#3	installation of sensors	done: M18
#4	Web-based registration of clouds and images	done: M18
#5	Web-based registration of application binaries	done: M18
#6	Flexiant driver for Cloudify	done: M18
#7	support for TSDB bootstrapping	done: M18
#8	CAMEL-based API	started, finishes <M24
#9	multi-cloud deployment	starts > M21
#10	integration of scalability rules	starts >M21
#11	installation of dedicate sensors	starts >M21
#12	support for deployment changes	starts >M21

	features of run-time entities	**status**
#13	basic exeution engine	done: M18
#14	more sophisticated execution engine	starts >M21
#15	interceptor implementation	starts >M21
#16	interpretor implementation	starts >M21
#17	migration support	starts >M30

	features of monitoring entities	**status**
#18	single-cloud scale up of components	done: M18
#19	support for multiple sensors	done: M18
#20	support for basic monitoring data	done: M18
#21	collect monitoring data at a single location	done: M18
#22	aggregation of monitoring data	done: M18
#23	simple scale out of application per cloud	done: M18
#24	feed monitoring data back to meta-data database	done: M18
#25	support for deployment changes	>M21
#26	normalisation of monitoring data	>M18, finishes <M24
#27	support scalability for monitoring infrastructure	>M21
#28	derive required sensors from scalability rules	>M24
#29	mechanisms to support Upperware by detecting service level objective violations	>M24

Figure 15: Overview of planned and realised features for the Executionware core entities.

	features for HyperFlow	status
#1	cloud-enablement of HyperFlow platform	done: M18
#2	cloud bootstrapping functionality	done: M18
#3	modelling of some use cases in CloudML	done: M18
#4	integration in Executionware monitoring infrastructure	>M21

	features for Scalarm	status
#5	Perl-based reimplementation of Simulation Manager	done: M18
#6	generation of Scalarm-specific monitoring streams	done: M18
#7	provisioning of CloudML description	started
#8	support for running Scalarm workers with OpenStack	started
#9	support for running Scalarm workers with Flexiant	started
#10	integration in Executionware monitoring infrastructure	>M21

Figure 16: Overview of planned and realised features for integration support for workflows and data farming applications.

2. Bootstrapping of a distributed time-series database instance per cloud
3. Be accessible by the Upperware through a HTTP-based API
4. Deploy single applications consisting of multiple components to a single cloud
5. Monitor the entire application and component instances based on sensors
6. Scale out components due to overload on a per-cloud basis
7. Collect and aggregate monitoring data
8. Feed back monitoring data to the meta-data database

Concluding, this functionality serves well as a solid prototype with a basic functionality, that yet offers flexibility and the possibility for step-by-step enhancements for more sophisticated functionality in later stages of the project.

The provisioning of two testbeds enables that Executionware features can be tested during the integration period M19–M21.

In parallel to the development of the first Executionware prototype, serious efforts have been made to prepare the integration of both workflow and data farming functionality in the PaaSage Executionware. Here, the initial work has been to make the used platforms, i.e. HyperFlow and Scalarm, cloud-enabled and support their deployment in cloud environments (cf. Figure 16). In particular, support for running them on Executionware's main cloud infrastructures OpenStack and Flexiant has been realised.

9.2 Roadmap to Final Version

Open issues and planned work can be found in any of the areas covered in this document. We sketch them following the structure used on this document.

Testbeds

While Flexiant's testbed is available and ready for used, there is ongoing work in GWDG's OpenStack platform. This work includes opening up OpenStack's Nova API access to clients outside the GWDG network. This is required by the consortium in order to enable automatic control of resources by PaaSage system components such as the Executionware. In addition, CETIC is planning to establish another OpenStack-based testbed running the latest OpenStack release Icehouse to be released in April 2014. Flexiant, in turn, targets the implementation of an open-source, client driver for their platform.

Infrastructure Entities

With respect to the infrastructure entities, future work will include the support and definition of a CAMEL-based API to the Executionware frontend. This includes support for deployment changes. Moreover, realising support for multi-cloud deployment and for extended scalability rules [4] are the natural next steps. In particular, the provisioning of sophisticated scalability rules enables sensor selection at run-time and deployment-time which may reduce the amount of overall monitoring data.

Further tasks include the integration of frontend data store with the meta-data database. Similar, the user interface for the registry shall be integrated with the social network. Both, meta-data database and social network are developed in work package 4 and described in deliverable D4.1.1 [5].

Run-time Entities

With respect to the run-time entities, the next steps include the provisioning of basic interceptor and interpreter implementations which are unavailable at M18. These efforts will be accompanied by the development of a more sophisticated execution engine whose functionality is beyond what Cloudify offers. Finally, at later stages of the project we target to realise cross-cloud migration support of running component instances.

Monitoring Entities

With respect to monitoring entities, the primary focus will be on integrating scalability rules. In particular, it is necessary to automatically create metrics depending on the requirements of the particular deployment information. Also, mechanisms have to be established that enable the Upperware to realise scalability across cloud boundaries. Finally, the scalability of the entire monitoring infrastructure has to be tackled. This is due to the fact that a single collector node may become overloaded when too many data providers are in the system.

Workflow Applications

The primary task of integrating the workflow engine with the Executionware is an integration and extension of the Executionware monitoring infrastructure such that custom, application-specific events can be *(i)* emitted, *(ii)* filtered, and *(iii)* trigger custom, application-specific actions.

For the integration phase (M19–M21), we attempt to realise a Cloudify-compatible version of the workflow engine that can be deployed using the M18 prototype. For that purpose, additional bootstrapping scripts will to be prepared.

Data Farming Applications

The integration of data farming application with our Executionware has to happen on the level of *application description*, *scalability*, and *API*:

A transparent deployment of Scalarm applications is only possible after each Scalarm service has been described with CloudML [3]. Only then will the Executionware be able to handle all actions required for the successful deployment of the whole Scalarm platform.

So far, we have evaluated the possibilities for generating Scalarm-specific monitoring streams. Scalarm will enable scalability by exploiting PaaSage's monitoring and scaling rules. We define a scalability rule for each Scalarm service based on monitoring parameters and taking into account non-functional parameters. As any PaaSage application component, Scalarm services generate

a stream of monitoring data that is received and processed by the monitoring infrastructure.

Making the Experiment Manager PaaSage-aware leverages the benefits of the API offered by the Interpreter entity. Such awareness enables the end user to manually increase the amount of computational resources dedicated to run an experiment as long as no global constraints (e.g., total costs) are violated.

References

[1] Bartosz Balis. “A model of computation and enactment engine for complex distributed workflows”. In: *EuroPar*. submitted. 2014.

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

[3] The PaaSage Consortium. *D2.1.1—CloudML Guide and Assessment Report*. PaaSage project deliverable. Oct. 2013.

[4] The PaaSage Consortium. *D2.1.1—CloudML Implementation Documentation—First Version*. PaaSage project deliverable. Mar. 2014.

[5] The PaaSage Consortium. *D4.1.1—Prototype Metadata Database and Social Network*. PaaSage project deliverable. Mar. 2014.

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

[7] Luca Deri, Simone Mainardi and Francesco Fusco. “Tsdb: A Compressed Database for Time Series”. In: *Proceedings of the 4th International Conference on Traffic Monitoring and Analysis*. TMA’12. Vienna, Austria: Springer-Verlag, 2012, pp. 143–156. ISBN: 978-3-642-28533-2. DOI: 10.1007/978-3-642-28534-9_16. URL: http://dx.doi.org/10.1007/978-3-642-28534-9_16.

[8] Steffen Kächele, Christian Spann, Franz J. Hauck and Jörg Domaschka. “Beyond IaaS and PaaS: An Extended Cloud Taxonomy for Computation, Storage and Networking”. In: *UCC 2013: IEEE/ACM 6th International Conference on Utility and Cloud Computing*. IEEE Computer Society, 2013.

[9] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine and Daniel Lewin. “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web”. In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. STOC ’97. El Paso, Texas, USA: ACM, 1997, pp. 654–663. ISBN: 0-89791-888-6. DOI: 10.1145/258533.258660. URL: http://doi.acm.org/10.1145/258533.258660.

[10] 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.

Appendix

In this appendix, we provide technical details to the individual software modules developed for the Executionware M18 prototype. Where applicable, we refer to sources of available (online) documentation. We consider the individual modules and applications one-by-one.

A Executionware Frontend

With respect to the Executionware frontend, this section details implementation aspects and licensing schema. Furthermore, it states more technical features of Cloudify and finally, it presents a sketch of the Executionware API as it will be used by the Upperware.

A.1 Implementation Aspects

The Web frontend has been implemented using the Play framework[8] and is ready for use. The data is currently stored in a PostgreSQL database but will be migrated to the meta-data database in later stages of the project. The description of how to install the Executionware (including the Web frontend) will be documented in the source code repository once the Executionware frontend has been released as open source software. The planned licensing scheme will be an Apache 2.0 license.

A.2 Cloudify Cloud Configuration

As stated in Section 4.1, each cloud known to a Cloudify installation has its own configuration directory that also defines the cloud's name. Within this directory resides a `.properties` file that defines environment variables and a `.groovy` file that contains the actual cloud definition including available virtual machine types (called templates), references to operating system images, the configuration for the management VMs, and the cloud driver to be used.

A.3 Cloudify Recipes

The Executionware frontend enables the deployment of Cloudify recipes. As discussed in Section 5.2 this requires that valid Cloudify recipes be registered at the Executionware frontend first.

[8] `http://www.playframework.com`

```
/
/<appname>-application.groovy     # application recipe
/<appname>-application.properties # application properties
/<service1>       # directory with installation and
                  # lifecycle scripts for service1
/<serivce...>     # other service directories
/<serviceN>       # directory with installation and
                  # lifecycle scripts for serviceN
/...              # other files required by the application
```

Figure 17: Archive file containing an application description

```
/
/<servname>-service.groovy      # service recipe
/<servname>-service.properties  # service properties
/<servname>_<lifecycl>.groovy   # service lifecycle handlers
/...                            # other files required by the
                                # recipe
```

Figure 18: Archive file containing a service description

Application recipes are stored as archive files. The structure of such a file is as indicated in Figure 17. The `/app` directory contains any data and files that are required to deploy the application including all of its services (cf. Section 4.1).

In order to re-use individual components, an application recipe may make use of a service recipe. Again, service recipes can registered as archive files through the Executionware frontend. The structure of such a file is as indicated in Figure 18.

A.4 CAMEL-based API

In the following, we develop an Executionware/Upperware API that functions as a starting point for the upcoming integration phase. The primary goal of the API is to provide *(a)* a deployment mechanism that is easy to use by the Upperware and *(b)* avoids duplicated functionality between Upperware and Executionware. In particular, it does not seem reasonable to have CloudML interpretation and parsing functionality available in the Executionware as well. In order to maintain the service characteristics of the Executionware, all operations are offered as a JSON/REST interface over the HTTP protocol.

HTTP op.	URI	description
GET	/cloud/	list all available clouds
GET	/cloud/$id	list details for this cloud
GET	/cloud?query	list available clouds with queried properties
GET	/machine/	list available machines
GET	/machine/$id	list details for this machine
GET	/machine?query	list available machines with queried properties
POST	/machine/	registers a new machine template (CPU, memory, storage configuration with a particular cloud). Returns a UUID for that machine.
DELETE	/machine/$id	deletes the particular machine template
GET	/image/	list all available images
GET	/image/$id	list all details for this image
GET	/image?query	list available images with queried properties

Figure 19: Part of Executionware API dealing with clouds, images, and virtual machine configurations

Dealing with Cloud Properties

In order to be able to deploy applications, it is necessary that the hardware properties of virtual machines, the images running on them, as well as the cloud they are acquired from be specified. Figure 19 lists the API for dealing with all those cloud-related aspects. The following paragraphs briefly discuss the respective aspects.

Cloud Definition As stated in Section 5 the Executionware frontend comes with a GUI-based mechanism to add clouds to a registry. In order to enable the Executionware to make use of this registry, the Upperware interface offers a mechanism to query and to list available cloud platforms.

Virtual Machine Configuration In addition to clouds, a IaaS-based application requires information about available hardware configurations. In particular, it should be possible to reference dedicated combinations of particular hardware combinations by a common identifier. For that reason, the API of the Executionware frontend enables the use of virtual machine templates that depict particular

machine properties. This enables the Upperware to register such properties with the Executionware and further re-use them as required.

Image Definitions Finally, a virtual machine instance requires an image that defines the file system visible by the operating system and the applications running in this virtual machine.

Dealing with Application Properties

With respect to applications we require a separate view on components, i.e. deployable artefacts, on the one hand. Each of these elements is accompanied by a set of files and life-cycle handlers. On the other hand, for deploying applications, components have to be grouped in multiple ways. First, they have to be assigned to applications and second, they have to be bundled according to their assignment to virtual machines. In the following, we first discuss operations to handle components, applications, and bundles (cf. Table 20). Then, we also consider life-cycle handlers and files (cf. Table 21). For describing the API, we overrule the definitions of the terminology section.

Components A component is a representative for an executable entity (e.g. a binary) that can be deployed and run. It can be enriched with various life-cycle handlers (see below) that are used at various stages of the component's life-cycle. For instance for installing the correct binaries. *Per se* components are fully application opaque. That is, they cannot be deployed as is, but have to be put in the context of an application, possibly by being grouped with other components. This is done through `bundles` (see below).

Applications An application defines a closure for a set of deployable and is the primary access point for starting and stopping the entire application. The application also defines how components are grouped (ie. which ones run on the same virtual machine and scale out together).

Bundles A bundle denotes a group of components shall shall be deployed together on a particular cloud. For that reason, bundles always depend on an application and further are self-contained in the sense that they can be scaled independent from other bundles.

Handlers A life-cycle handler is an executable entity (e.g. a binary or a script) whose purpose is to deal with a particular event at a component's life-cycle. Handlers are dedicated to a particular component. Yet, a component may have multiple handlers for the same event, but different operating systems.

HTTP op.	URI	description
GET	/component/	list all available components
GET	/component/$id	list details for this component
GET	/component?query	list all available components with queried properties
POST	/component/	add a new component to the registry
PUT	/component/$id	change this component accordingly
DELETE	/component/$id	delete component and all of its handlers
GET	/application/	list all available applications
GET	/application/$id	list details for this application
GET	/application?query	list all available applications with queried properties
POST	/application/	add a new application to the registry
DELETE	/application/$id	delete application
GET	./bundle/	list all available bundles
GET	./bundle/$id	list details for this bundle (e.g. #instances)
GET	./bundle?query	list all available bundles with queried properties
POST	./bundle/	add a new bundle to the registry
DELETE	./bundle/$id	delete bundle undeploying all of its instances

Figure 20: Part of Executionware API dealing with binaries, recipes and application deployment. The root for the bundle URIs are `/application/$id`. The bundle information contains for instance the number of currently running instances.

HTTP op.	URI	description
GET	./handler/	lists all available lifecylce handlers registered with that component
GET	./handler?query	list details for this handler
POST	./handler/	add a new handler to that component
DELETE	./handler/$id	delete this handler
GET	./file/	list all files registered with this component
DELETE	./file/$id	delete this file
POST	./file/	create new file (uploads the binary)
PUT	./file/$id	modify this file (uploads a new binary)

Figure 21: Part of Executionware API dealing with handlers and additional files. The root for each URI is a dedicated component: `/component/$id`.

File A file is a named sequence of binary data. All files registered with a component will be copied to any location where this component is deployed.

B Time-series Database and Monitoring

KairosDB has already been installed and configured on three Flexiant FCO VMs. The following URL directs to the web interface of one KairosDB installation (in one of the three VMs), while the other two installations are command line based.

`http://109.231.122.119:8080/`

Finally, source code for inserting data in a KairosDB instance (i.e. a KairosDB client), as well as the implementation of the Metrics Collector component that mediates between the MDDB and KairosDB will be provided before M18.

C Workflow Platform

Installation instructions for HyperFlow[9] and the AMQP executor[10] are available online together with the current version of recipes for HyperFlow deployment[11]. In the following, we present some technical details on the respective deployment recipes as well as an example for configuring an for being used with HyperFlow.

[9] `https://github.com/dice-cyfronet/hyperflow`
[10] `https://github.com/dice-cyfronet/hyperflow-amqp-executor`
[11] `https://github.com/malawski/hyperflow-deployment`

C.1 Provided Cookbook

The cookbook for deployment of HyperFlow modules and applications is designed to be used with `chef-solo`, so no installation or access to chef-server is required. For the M18 prototype, the following recipes have been developed.

HyperFlow Engine This recipe installs the current release of HyperFlow engine by downloading it from GitHub[12]. It installs the `node.js` runtime using a third-party `node.js` recipe, the `Redis` database package as well as all packages `Redis` depends on. For the `Redis` installation, the NPM package manager of `node.js` is used. An additional recipe is used to install and configure the `RabbitMQ` message broker for AMQP communication.

AMQP Executor This recipe installs the current release of AMQP executor of HyperFlow. As dependencies it installs Ruby 2.0. The executor itself is a installed as Ruby Gem package, together with its dependencies. The recipe creates also startup scripts for cloud-init configuration system and optionally the configuration of credentials for accessing storage for the executor.

Molecular docking This recipe installs USTUTT's Molecular Docking application by building it from its source code. It also installs the depending packages such as the `OpenMPI`[13] library and the *gcc* compiler. Moreover, further recipes have been created in order to install and configure additional package dependencies such as *The Persistence of Vision Raytracer (POV-Ray)*[14] for converting the output results into image files, and the `lighttpd`[15] web server for downloading the output results and the image files.

C.2 Usage Example

All the recipes used by the workflow engine can be grouped in the configuration files for the specific VMs. Examples of such configuration for Master and Worker nodes are given in Figure 22.

In order to deploy the molecular docking application provided by USTUTT using HyperFlow, a JSON description file is created, as shown in Figure 23. From this figure, the application is composed of three stages or HyperFlow processes that need to be run in a sequential order, i.e. *(i)* pre-processing; *(ii)* running of the application (which includes compiling the source code), and *(iii)* post-processing.

[12] `https://github.com/`
[13] `http://www.open-mpi.org/`
[14] `http://www.povray.org/`
[15] `http://www.lighttpd.net/`

```
// master
// include recipes in the runlist
   "run_list": [
      "recipe[nodejs::default]",
      "recipe[rabbitmq::default]",
      "recipe[rabbitmq::mgmt_console]",
      "recipe[workflows::hyperflow-engine]"
   ]
}

{
// worker
// set node attributes
//  "amqp_executor": {
//    "AMQP_URL" : "amqp://login:password@host
    :5672/vhost",
//    "AWS_ACCESS_KEY_ID" : "XXXXXXXXXXX",
//    "AWS_SECRET_ACCESS_KEY" : "
    YYYYYYYYYYYYYYYYYYYYYYYYY"
//  },
// include recipes in the runlist
   "run_list": [
      "recipe[workflows::molecular_docking]",
      "recipe[workflows::povray]",
      "recipe[workflows::amqp-executor]",
      "recipe[workflows::lighttpd]"
   ]
```

Figure 22: Example of configuration scripts for deployment of HyperFlow Master and Worker nodes.

In the *pre-processing* stage, a universally unique identifier (UUID) is generated for each workflow session in order to prevent the results being overwritten, as the MD application may run several different workflow instances with varying input parameters. Morevoer, a new directory based on this UUID is automatically created and linked to the web server, so the user will be able to download the results from the web browser. Then, this UUID is being passed to the next stages in the workflow, where in the *running* stage, the script uploads the MD source code to this UUID directory, compiles the code, and runs the program. In Fig. 23 line 32, the workflow needs to be run on 8 cores for $1,000$ molecules until 2 simulation end time. However, other workflows may have different parameters, as mentioned earlier.

In the final *post-processing* stage, the output results are being converted to image files by the POV-Ray library. Then, these output and image files are bundled together as a compressed file, where it is available to be downloaded by the user.

```
{
    "name": "cmd",
    "functions": [ {
      "name": "md_preprocess",
      "module": "functions/molecular_docking"
    }, {
      "name": "md_run",
      "module": "functions/molecular_docking"
    }, {
      "name": "md_postprocess",
      "module": "functions/molecular_docking"
    } ],
    "processes": [ {
        "name": "pre-processing",
        "type": "dataflow",
        "function": "md_preprocess",
        "config": {
            "executor": {
                "executable":
                    "/home/paasage/script/pre-processing.sh",
                "args": ""
            }
        },
        "ins": [ "start" ],
        "outs": [ "pre", "dir_uuid" ]
    }, {
        "name": "running",
        "type": "dataflow",
        "function": "md_run",
        "config": {
            "executor": {
                "executable":
                    "/home/paasage/script/run-program.sh",
                "args": "8 1000 2"
            }
        },
        "ins": [ "pre", "dir_uuid" ],
        "outs": [ "run" ]
    }, {
        "name": "post-processing",
        "type": "dataflow",
        "function": "md_postprocess",
        "config": {
            "executor": {
                "executable":
                    "/home/paasage/script/post-processing.sh",
                "args": ""
            }
        },
        "ins": [ "run", "dir_uuid" ],
        "outs": [ "post" ]
    } ],

    ...
}
```

Figure 23: A snippet of the JSON description for the Molecular Docking application.