



# ARCADIA

A novel reconfigurable by design highly distributed applications  
development paradigm over programmable infrastructure

---

## Deliverable D4.2

### Description of the Applications' Lifecycle Management Support

---

<b>Editor(s):</b>	Tran Quang Thanh (TU-Berlin), Stefan Covaci (TU-Berlin) Panagiotis Gouvas (UBITECH)
<b>Responsible Organization(s):</b>	Technische Universität Berlin
<b>Version:</b>	1.0
<b>Status:</b>	Final
<b>Date:</b>	30 April 2016
<b>Dissemination level:</b>	Public

## Deliverable fact sheet



<b>Grant Agreement No.:</b>	645372
<b>Project Acronym:</b>	ARCADIA
<b>Project Title:</b>	A novel reconfigurable by design highly distributed applications Development paradigm over programmable infrastructure
<b>Project Website:</b>	<a href="http://www.arcadia-framework.eu/">http://www.arcadia-framework.eu/</a>
<b>Start Date:</b>	01/01/2015
<b>Duration:</b>	36 months

<b>Title of Deliverable:</b>	D4.2 Description of the Applications' Lifecycle Management Support
<b>Related WP:</b>	WP4 – ARCADIA Development Toolkit
<b>Due date according to contract:</b>	31/03/2016

<b>Editor(s):</b>	Tran Quang Thanh (TU-Berlin), Stefan Covaci (TU-Berlin) Panagiotis Gouvas (UBITECH)
<b>Contributor(s):</b>	Anastasios Zafeiropoulos (UBITECH) Alessandro Rossini (SINTEF) Nikos Koutsouris (WINGS)
<b>Reviewer(s):</b>	Matteo Repetto (CNIT) Lukasz Porwol (NUIG)
<b>Approved by:</b>	All Partners

<b>Abstract:</b>	This deliverable will provide a description of the software development methodology that to be followed for deploying highly distributed applications based on the ARCADIA's components.
<b>Keyword(s):</b>	<i>Application Lifecycle Management, DevOps, Microservice, Orchestration</i>

## Partners

 <p>NUI Galway OÉ Gaillimh</p> 	<p>Insight Centre for Data Analytics, National University of Ireland, Galway</p> <p>Stiftelsen SINTEF</p>	<p>Ireland</p> <p>Norway</p>
	<p>Technische Universität Berlin</p>	<p>Germany</p>
	<p>Consorzio Nazionale Interuniversitario per le Telecomunicazioni</p>	<p>Italy</p>
<p>Univerza v Ljubljani</p> 	<p>Univerza v Ljubljani</p>	<p>Slovenia</p>
	<p>UBITECH</p>	<p>Greece</p>
	<p>WINGS ICT Solutions Information &amp; Communication Technologies EPE</p>	<p>Greece</p>
	<p>MAGGIOLI SPA</p>	<p>Italy</p>
	<p>ADITESS Advanced Integrated Technology Solutions and Services Ltd</p>	<p>Cyprus</p>

## Revision History

Version	Date	Editor(s)	Remark(s)
0.1	01/03/2016	Tran Quang Thanh (TUB) Stefan Covaci (TUB)	Defining ToC
0.2	27/03/2016	Panagiotis Gouvas (UBITECH) Anastasios Zafeiropoulos (UBITECH) Alessandro Rossini (SINTEF) Nikos Koutsouris (WINGS) Matteo Repetto (CNIT) Tran Quang Thanh (TU-Berlin) Stefan Covaci (TU-Berlin)	First version with inputs from partners
0.3	30/03/2016	Panagiotis Gouvas (UBITECH) Anastasios Zafeiropoulos (UBITECH) Alessandro Rossini (SINTEF) Nikos Koutsouris (WINGS) Matteo Repetto (CNIT) Tran Quang Thanh (TU-Berlin) Stefan Covaci (TU-Berlin)	Additions to All Chapters
0.8	28/04/2016	Tran Quang Thanh (TU-Berlin) Panagiotis Gouvas (UBITECH)	Consolidated version of all chapters
1.0	30/04/2016	Tran Quang Thanh (TU-Berlin) Anastasios Zafeiropoulos (UBITECH)	Updating reference and finalizing

### Statement of originality:

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

## Executive Summary

This report describes a novel software development and operation methodology to be followed for developing and operating highly distributed applications (HDA). In ARCADIA, a HDA is an application deployed on cloud infrastructures and delivered as services. Specifically, different application lifecycle management (ALM) phases have been taken into account including microservices development, service graph composition, deployment, and orchestration.

The adopted methodology is in line with the current development of software industry (e.g. microservice pattern design, DevOps) as well as related standards and research works (e.g. TOSCA NFV specification). Several components are currently under development –constituting the overall ARCADIA framework–aiming to support software developers either implementing an ARCADIA HDA from scratch (native approach) or leveraging existing solutions (adaptation of legacy applications). Apparently, native ARCADIA application will benefit from the full set of capabilities of the framework.

In addition to that, this deliverable will elaborate on the microservice metamodel that the ARCADIA development supports and will discuss implementation issues regarding the actual ALM features supported by the developed platform.

# Table of Contents

EXECUTIVE SUMMARY .....	5
TABLE OF CONTENTS .....	6
LIST OF FIGURES.....	7
<b>1 INTRODUCTION .....</b>	<b>8</b>
1.1 PURPOSE AND SCOPE .....	8
1.2 RELATION WITH OTHER WPS .....	8
<b>2 HIGHLY DISTRIBUTED APPLICATION .....</b>	<b>9</b>
2.1 ARCADIA HDA TYPES.....	9
2.2 ARCADIA HDA LIFECYCLE .....	9
2.3 ARCADIA MICROSERVICE METAMODEL.....	11
2.4 ARCADIA METAMODEL VS. TOSCA.....	13
<b>3 DEVELOPMENT OF MICROSERVICES.....</b>	<b>14</b>
3.1 ARCADIA WEB-BASED IDE ENVIRONMENT .....	14
3.2 ARCADIA IDE PLUG-IN .....	15
3.2.1 ARCADIA IDE plug-in.....	16
3.2.2 Interaction with the Smart Controller .....	18
3.3 ARCADIA COMPONENT VALIDATION & STORAGE .....	19
3.4 ARCADIA COMPONENT BUNDLING.....	21
<b>4 SERVICE GRAPH COMPOSITION .....</b>	<b>23</b>
4.1 CREATION OF SERVICE GRAPHS .....	23
4.2 POLICY MANAGEMENT AT THE SERVICE GRAPH LEVEL.....	26
<b>5 HDA DEPLOYMENT AND ORCHESTRATION .....</b>	<b>29</b>
5.1 DEPLOYMENT POLICIES.....	29
5.2 SERVICE GRAPH GROUNDING.....	29
5.3 RUNTIME ORCHESTRATION.....	30
<b>6 CONCLUSIONS.....</b>	<b>34</b>
<b>ACRONYMS .....</b>	<b>35</b>
<b>REFERENCES .....</b>	<b>36</b>

## List of Figures

Figure 1: HDA Lifecycle .....	10
Figure 2: Generation of a new API key.....	15
Figure 3: ARCADIA plug-in context menu in the Eclipse Che IDE .....	16
Figure 4: Available Annotations.....	17
Figure 5: Development of ARCADIA component through web-based IDE.....	18
Figure 6: Maven module that performs the introspection .....	19
Figure 7: Serialization Model for a specific component .....	20
Figure 8: Components' View from the dashboard .....	21
Figure 9: Service Graph Composition .....	24
Figure 10: Serialization Model for a Service Graph .....	25
Figure 11: Overview of Service Graphs from ARCADIA Dashboard .....	26
Figure 12: Creation of Runtime Policies.....	27
Figure 13: Serialization Model for a Grounded Service Graph.....	30
Figure 14: Component's States during deployment.....	32
Figure 15: Orchestrator's States during deployment .....	33

# 1 Introduction

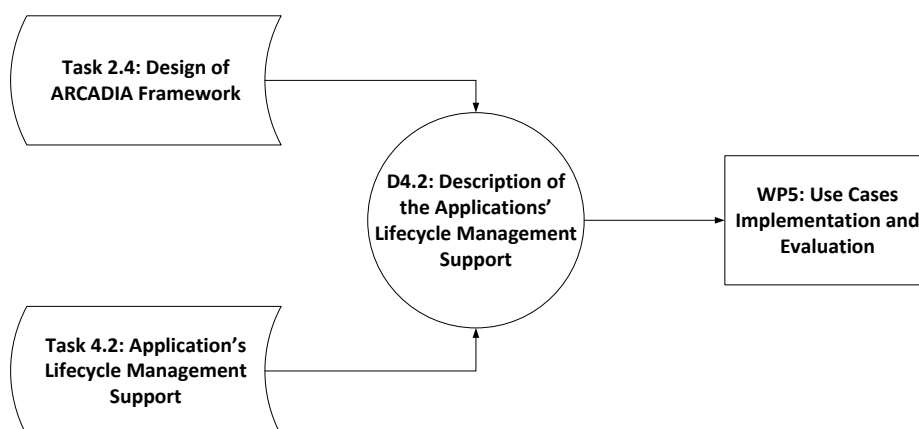
## 1.1 Purpose and Scope

This document provides a description regarding the lifecycle of Highly Distributed Applications (**HDA**). The definition of an HDA application in the frame of ARCADIA has been already provided in the ARCADIA Architecture deliverable [1]. However, in the frame of this document, the actual characteristics that an HDA should have in order to be managed are thoroughly discussed. HDA management in ARCADIA refers to the overall Application Lifecycle Management (from now on **ALM**) which is divided into several distinct phases [2]. In ARCADIA, applications are being distributed, assembled from many small components (also addressed as microservices), developed by distributed teams, and deployed on different cloud infrastructures. As a result, HDA applications are practically service graphs consisting of components that satisfy a specific metamodel. In the context of ARCADIA project, the primary focus is put into the development and operation aspects of ALM to support multi-cloud environments.

Towards the above lines, this deliverable will elaborate on the HDA metamodel that the ARCADIA development supports and will discuss implementation issues regarding the actual ALM features supported by the developed platform.

## 1.2 Relation with other WPs

The relationship of D4.2 with other tasks and WPs is shown below. This deliverable is the outcome of the tasks “Task 4.2: Applications' Lifecycle Management Support” of WP4. The aim of this task is to analyze and describe the different lifecycle phases of a Highly Distributed Applications which are briefly outlined in Task 2.4 about ARCADIA framework design and are going to be followed in WP5 (ARCADIA use cases implementation and evaluation). The covered phases include the microservices development, the service graph composition, the deployment, and the orchestration.





## 2 Highly Distributed Application

A Highly Distributed Application (HDA) is defined as a distributed scalable structured system of software entities implemented to run on a multi-cloud infrastructure. The structure of an HDA enables high flexibility in reusing and composing basic services, like databases, web front-ends, authentication modules, network functions and so on. Each component comes with its configuration and deployment requirements, and the logical process of chaining the components together must be defined. The implementation of the HDA concept needs the interaction among different business roles and the presence of underlying physical and virtual resources.

### 2.1 ARCADIA HDA Types

In ARCADIA, we support the deployment and operation of:

**ARCADIA applications:** applications that are going to be developed following the proposed software development paradigm. There are three supported types of applications:

- Native ARCADIA applications: will benefit from the full set of capabilities of the framework
- Legacy applications: existing applications already available in an executable form
- Hybrid applications: applications that consist of application tiers from both the cases above

Regarding Legacy applications, a subset of offered capabilities would be possible to be available. For a chain of legacy application tiers to be deployable and benefit from the ARCADIA framework, proper interfacing should be built between application tiers while appropriate annotations could accompany the executables. In fact, each legacy executable should be interfaced by developing an ARCADIA Binding Interface (**BI**), which will expose its functionalities and enable communication with other appropriately enhanced legacy executables in a common ARCADIA framework style/way. The same stands for hybrid applications as it concerns the legacy application tiers; an ARCADIA BI exposes its functionalities and enables communication with legacy or native application tiers while proper annotations accompany the executable. In order a legacy application tier or legacy application as a chain to be usable by an ARCADIA application tier at the time of development, it should be enhanced with an ARCADIA BI. It is a manual operation performed by the DevOps. It should be noted that for native ARCADIA applications, the supported BIs are automatically included in the development phase, based on the proposed software development paradigm.

### 2.2 ARCADIA HDA Lifecycle

The transition to the microservice development model revolutionized the way software architects design, build and ship software. Many frameworks have been introduced (e.g. Spring-boot [3]) that allow developers to create their services faster through the avoidance of boilerplate code. In parallel, individual

microservices are rarely deployed in a standalone manner since only through the collaboration of specific microservices complex applications can be created. To this end, complex applications **constitute practically direct acyclic graphs of dependencies**. These acyclic graphs will be hereinafter addressed as **service graphs**.

Figure 1 summarizes the discrete phases that are involved in the development of HDAs. More specifically, these phases include the individual development of microservices (made available in the microservices repository), the composition of service graphs (through the DevOps toolkit – made available in the service graph repository), the initialization/parameterization of service graphs (through the DevOps toolkit) and the multi-cloud deployment of service graphs (deployment over programmable infrastructure in an operational environment). Multi-cloud deployment of service graphs is taking into account the defined policies on behalf of the service provider, supporting in this way optimal placement of a service graph over programmable infrastructure as well as the application of runtime policies for dynamically adapting the execution context based on a set of predefined rules and actions.

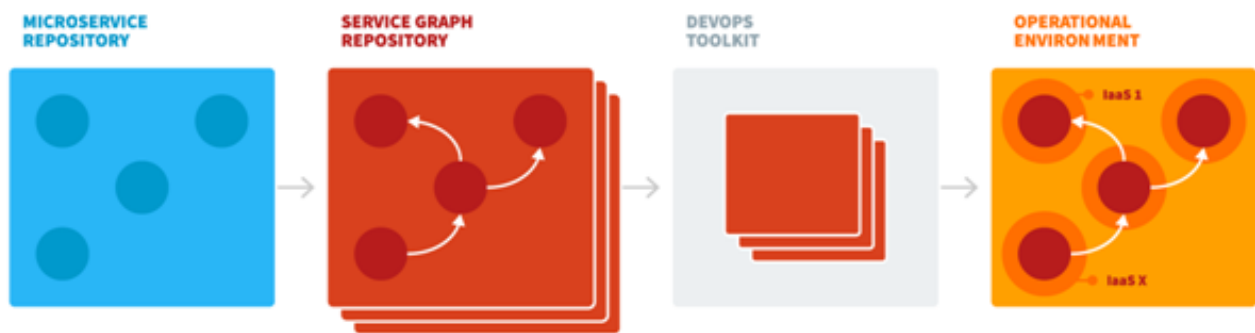


Figure 1: HDA Lifecycle

Although the microservice paradigm has prevailed, there is a fundamental question regarding microservices: “**what is actually an ARCADIA microservice?**” Is it a service that can just be reused by other services (like the plain old web services that prevailed in the SOA era)? Is it a service that has a small footprint regarding size (hence, the prefix micro is used)? In other words, it is extremely crucial to identify the basic principles that a developed service should have in order to be characterized as microservice.

The answer to this question may be surprising but is very simple. **There is no globally acceptable definition of the term microservice.** On the contrary, there is a huge debate about whether or not the term itself has significant value. On top of that, the emergence of programmable infrastructure added additional parameters that should be taken into consideration during a ‘strict’ definition of a microservice. The programmable infrastructure allows the dynamic reconfiguration of provisioned resources (VCPUs, memory, storage, bandwidth, security groups, etc.) which are capabilities that are rarely taken into consideration during microservice development. To this end, for the sake of common understanding,

ARCADIA formulated the definition of microservices-based on a common set of characteristics, as detailed in the following subsection.

## 2.3 ARCADIA Microservice Metamodel

In the frame of ARCADIA any microservice should:

- a. expose its **configuration parameters** along with their metadata (e.g. what the acceptable values? can the parameter change during the execution?);
- b. expose **chainable interfaces** which will be used by other microservices in order to create a service graph;
- c. expose **required interfaces** which will also be used during the creation of a service graph;
- d. expose quantitative metrics regarding the QoS of the microservice;
- e. encapsulate a **lifecycle management programmability** layer which will be used during the placement of one service graph in the infrastructural resources;
- f. be **stateless** in order to be **horizontally scalable by design**;
- g. be **reactive to runtime modification of offered resources** in order to be **vertically scalable by design**;
- h. be **agnostic to physical storage, network, and general purpose resources**.

Regarding **(a)**, it could be argued that if a microservice entails a specific configuration layer, it is extremely crucial to be reconfigurable-by-design i.e. to adapt to the new configuration without interrupting its main thread of execution. Imagine a scenario of a video transcoder which is exposed as a microservice. If during a transcoding session the resolution of the video receiver is downgraded, the transcoding service should (upon request) adapt to the new 'mode' without the termination of the session. Regarding **(b)** and **(c)**, it is clear that dynamic coupling of services is highly valuable only when an actual binding can be fully automated during runtime. This level of automation raises severe prerequisites for the developed microservices. The 'profile' of the chaining should be clearly abstracted. Such a profile includes the offered/required data type, the ability to accept more than one chaining, etc. These metadata are often stored in high-efficient key-value stores (such as consul [4]) in order to be queried by requesting microservices. In the aforementioned transcoding example, such dynamic microservice lookup could be performed by the transcoding microservice in order to identify an object storage microservice which has some constraints e.g. a security constraint (to support seamless symmetric encryption) or a location constraint (e.g. the location of the storage service to be near the location of the transcoding service so as to minimize the storage delay).

Regarding **(d)**, it should be noted that, while microservice-agnostic metrics are easily measured, the quantification of business-logic-specific metrics cannot be performed if a developer does not implement specific application-level probes. Indicatively, following the transcoding example, the microservice-agnostic

metrics such as VCPU, memory utilization, and network throughput can be extracted through the instrumentation of the underlying IaaS by making use of its programmability layer. However, metrics such as the sessions that are actively handled, the average delay per each transcoding sessions, etc. cannot be quantified if the microservice developer does provide a thread-safe interface which reports the measurement of these metrics.

Regarding **(e)**, the recent developments in the virtualization compendium provided management capabilities that could not be supported in the past. For instance, the live migration from one hypervisor to another one is now integrated as a built-in core feature of KVM [5] for more than a year. Hence, a microservice that is running on a specific infrastructure may be literally ‘transported’ to another one without downtime. Yet the microservices dependencies may be affected by this choice. Imagine the storage service that the transcoding relies on, to be seamlessly migrated from Ireland to the USA within the same IaaS. This could violate some chaining constraint (e.g. delay, legal aspects). As a result, both microservices should expose a basic programmability layer which handles the high-level microservice lifecycle (e.g. remove-chained dependency).

Regarding **(f)**, any service that is stateless can scale easily with the usage of some ‘facilitating’ services such as network balancers or web balancers. Historically, these services were statically configured by administrators or by DevOps engineers. The emergence of the programmability infrastructure model will progressively ‘offload’ this task to Virtualized Functions (a.k.a. VFs) that are controlled by a cloud orchestrator. Ensuring the stateless behavior of a service graph is a challenging task since the entire business logic should entail stateless behavior in order to be horizontally scalable by design.

Regarding **(g)**, taking into consideration the developments in hypervisor technologies and OS kernels the last two years, it could be argued that that the barriers to dynamic provision and de-provision of resources on operating systems have been raised. However, the dynamic provisioning of resources to a virtualized system does not imply that these resources are automatically binded to the hosted microservice. On the contrary, in most of the times, the microservice has to be (gracefully) restarted in order to make use of the new resources.

Finally, regarding **(h)**, it should be noted that not every valid microservice is capable of being ported in a modern infrastructure. For example, imagine a Java developer that uses a file-system for persistency in the frame of the development of one microservice. This is considered anti-pattern extremely since the microservice cannot be hosted in any Platform-as-a-Service provider.

**ARCADIA framework provides a holistic framework that assists developers in implementing microservice-based complex applications supporting the development, composition, deployment and management of HDAs that consist of these microservices.**

## **2.4 ARCADIA Metamodel vs. TOSCA**

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [6] is a specification developed by the Organization for the Advancement of Structured Information Standards (OASIS). TOSCA provides a language for specifying the components comprising the topology of cloud-based applications along with the processes for their orchestration. TOSCA supports the specification of types, but not instances, in deployment models. In contrast, the ARCADIA metamodel [7] supports the specification of both types and instances. Therefore, in its current form, TOSCA can only be used at design-time, while the ARCADIA meta- model can be utilized at both design-time and run-time.

As part of the joint standardization effort of ARCADIA, MODAClouds [8], and PaaSage [9], SINTEF presented the models@run-time approach to the TOSCA technical committee (TC) and proposed to form an ad hoc group to investigate how TOSCA could be extended to support this approach. The TC welcomed this proposal and, on 3 September 2015, approved by unanimous consent the formation of the Instance Model Ad Hoc group. This group is led by Alessandro Rossini and Sivan Barzily from GigaSpaces and it is meeting online on a weekly schedule. This will guarantee that the contribution of ARCADIA will partly be integrated into the standard.

### 3 Development of Microservices

As already mentioned the first phase regarding the lifecycle management of HDAs regards to the development of the microservices that adhere to the ARCADIA metamodel (see 2.3). ARCADIA is designed to help developers create and deploy applications quicker and more efficiently. It automates many tasks such as code and annotation validation, application deployment and application monitoring. Moreover, ARCADIA framework is responsible for managing and orchestrating components, validating the connection between them and generating separately deployable artifacts (bundled as unikernels) for each component.

The development of microservices is performed using the **ARCADIA web-based IDE environment**. Through this environment, developers can use specific annotations that are validated by the smart controller and are automatically interpreted to orchestratable components.

#### 3.1 ARCADIA web-based IDE environment

To develop an ARCADIA application, the **ARCADIA IDE platform** has to be used. The IDE environment relies on a state-of-the-art web-based IDE called “Eclipse Che” [10], which gives the possibility to the developers to develop microservices using the Bring-Your-Own-Device (BYOD) principle i.e. without the need to setup anything locally. This was a core internal decision of the project since we consider that the concept of “Build-Ship-Run” that has been introduced by Docker [11] should be followed by ARCADIA. However, Eclipse Che is a general-purpose development environment. As such, it can be used to develop anything. However, we are only interested in the development and validation of ARCADIA-compliant microservices. Therefore, a specific plug-in has been developed that interacts with the Smart Controller in order to check the validity of developed microservices.

First, developers have to register with the ARCADIA platform (not publicly available at this point), which is the central point of the ARCADIA applications' ecosystem and the basis for creating applications according to the ARCADIA development paradigm. After the account is activated, developers can access the ARCADIA dashboard, which is the entry point for the ARCADIA Platform. In the dashboard, developers are presented with an overview of their account, and they can manage and configure their ARCADIA applications, create, modify and delete policies and have a full control of their ARCADIA account. Moreover, they can generate and revoke API keys, which are used for authentication purposes, and eliminate the risk of sending usernames and passwords over unsecured networks. A new API key must be generated to use the ARCADIA IDE plug-in. This process is depicted in Figure 2, which shows the API-key generation process as it is developed in the current version of the ARCADIA Dashboard.

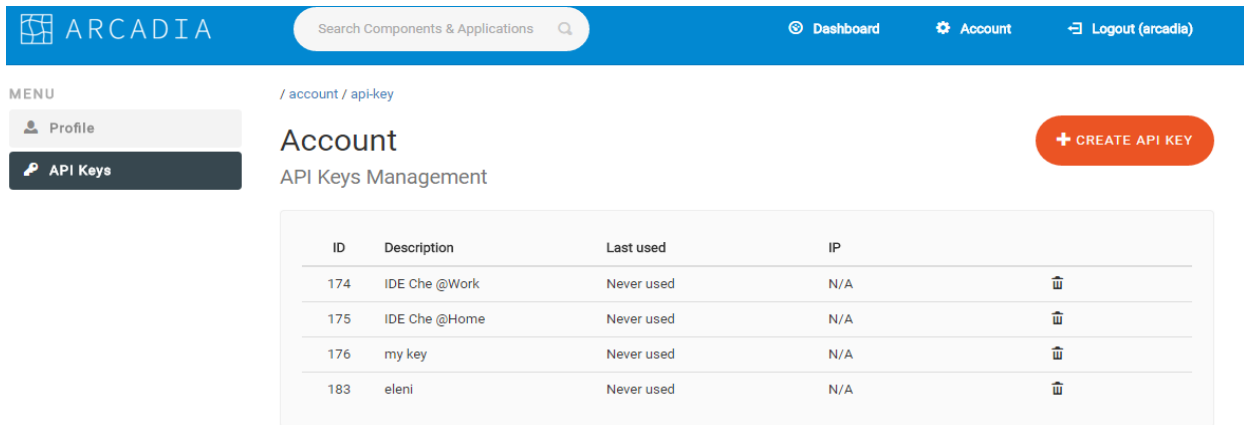


Figure 2: Generation of a new API key

### 3.2 ARCADIA IDE plug-in

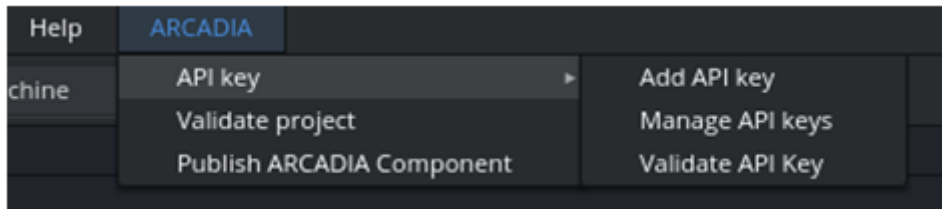
ARCADIA IDE plug-in is a vital part of the ARCADIA web-based IDE. It gives developers a way to communicate with the ARCADIA platform, without the need of leaving their IDE editor. Through the plug-in, developers can **a) manage their previously generated API keys, b) have a pre-compile validation of the ARCADIA annotations** and **c) submit their code to the platform.**

ARCADIA IDE plug-in is integrated with the latest version of Eclipse Che [9], which is the browser-based, cloud version of the classic Eclipse. Developers have to download and deploy Eclipse Che and install the ARCADIA IDE plug-in, after generating a new API key following the steps described in the previous section. Plug-ins for more IDEs will be available in the future. The ARCADIA IDE plug-in adds an “ARCADIA” menu entry on the top menu of the editor UI with every possible action provided by the plug-in. The final step of the setup is to configure the ARCADIA IDE plug-in with their generated API key. To do that, they can choose “Add API key” from the ARCADIA menu. The API key will be validated with the ARCADIA platform, and if it is valid, developers can start writing their ARCADIA application.

Eclipse Che is under extremely active development; hence, the ARCADIA plug-in is following along. Che is based on Docker [11], GWT [12], Orion [13] and RESTful APIs and offers both client and server side capabilities. The client-side of Che is based on the Orion editor, and it offers most of the features expected from a classic IDE such as automatic code completion, error marking, JAVA “intelligence” and documentation. On the other hand, the server side is responsible for managing the code repository, compiling and executing programs and managing runtimes. For similar reasons, ARCADIA IDE plug-in is also separated into two different modules. Both ARCADIA IDE plug-in and “Eclipse Che” are using maven for their builds.

### 3.2.1 ARCADIA IDE plug-in

The ARCADIA client-side plug-in appears as a top menu entry in web-based editor. Its primary task is to assist developers during component development. Through the plug-in, developers can view all available ARCADIA annotations, including some Javadoc-based documentation, authenticate with the ARCADIA platform and trigger component validation, through the ARCADIA server-side plug-in (Figure 3).



*Figure 3: ARCADIA plug-in context menu in the Eclipse Che IDE*

Specifically, when a developer starts writing an ARCADIA annotation, the plug-in **offers auto-complete suggestions where developers can just choose which annotation they want to use**. In addition, they can read the specifications of each annotation, like the required fields, naming conventions and examples. Moreover, if they choose to use one of the provided component templates, the plug-in will manage any ARCADIA requirement like maven dependencies.

Also, developers can add the API key generated from the ARCADIA platform and maintain old ones. To check the validity of the key, the plug-in posts to the ARCADIA platform a request with an “Authorization HTTP header” that includes such key. The server sends back a JSON response with details about the key, as its validity or possible errors. If the key is valid, it is saved in the IDE editor using HTML Local Storage. For security reasons and to avoid cross-origin HTTP requests, the actual request is sent through the ARCADIA server-side plug-in with the help of a GWT RPC interface [12]. In addition to that, developers can also get a list of all their active API keys, and they can delete those that are not needed anymore. The fetch and delete flow follow a similar pattern using the server-side plug-in as a proxy for security reasons.

Using the web-based IDE, the developers create microservices that adhere to the ARCADIA component metamodel. This compliance is achieved through the usage of specific annotations that allow developers to declare components, register configuration parameters, define chainable endpoints, etc. An overview of the annotations that are available at present is shown in Figure 4. The detail information is further discussed in the Deliverable 4.1 [14]



ubitech / **arcadia-framework** PRIVATE

Unwatch 29 Star 0 Fork 0

Code Issues 7 Pull requests 0 Wiki Pulse Graphs Settings

Branch: master

New file Upload files Find file History

**arcadia-framework** / annotation-libs / src / main / java / eu / arcadia / **annotations** /

nlykousas Annotation Interpreter + agentJson Helpers		Latest commit 96a6b62 15 days ago
..		
ArcadiaComponent.java	Changed XSD Model to facilitate Configuration and Metric persistency	2 months ago
ArcadiaConfigurationParameter.java	Runtime introspection	2 months ago
ArcadiaConfigurationParameters.java	Runtime introspection	2 months ago
ArcadiaMetric.java	Runtime introspection	2 months ago
ArcadiaMetrics.java	Runtime introspection	2 months ago
DependencyBindingHandler.java	<a href="#">Annotation Interpreter + agentJson Helpers</a>	15 days ago
DependencyExport.java	Annotation Interpreter + agentJson Helpers	15 days ago
DependencyExports.java	Annotation Interpreter + agentJson Helpers	15 days ago
DependencyResolutionHandler.java	Annotation Interpreter + agentJson Helpers	15 days ago
LifecycleInitialize.java	Annotation Interpreter + agentJson Helpers	15 days ago
LifecycleStart.java	Annotation Interpreter + agentJson Helpers	15 days ago
LifecycleStop.java	Annotation Interpreter + agentJson Helpers	15 days ago
ParameterType.java	Added proper annotations that have to be handled by the introspection...	2 months ago
ValueType.java	Added proper annotations that have to be handled by the introspection...	2 months ago

Figure 4: Available Annotations

The usage of annotations is performed in a seamless way through the development environment. Figure depicts the development of an actual microservice. As it is depicted, the developer is using several annotations such as *@ArcadiaComponent*, *@ArcadiaMetric*, *@ArcadiaExport*, etc. The definition of these annotations during development is assisted by the IDE while the functional usage of these annotations takes place after the submission of the executable to the Smart Controller.

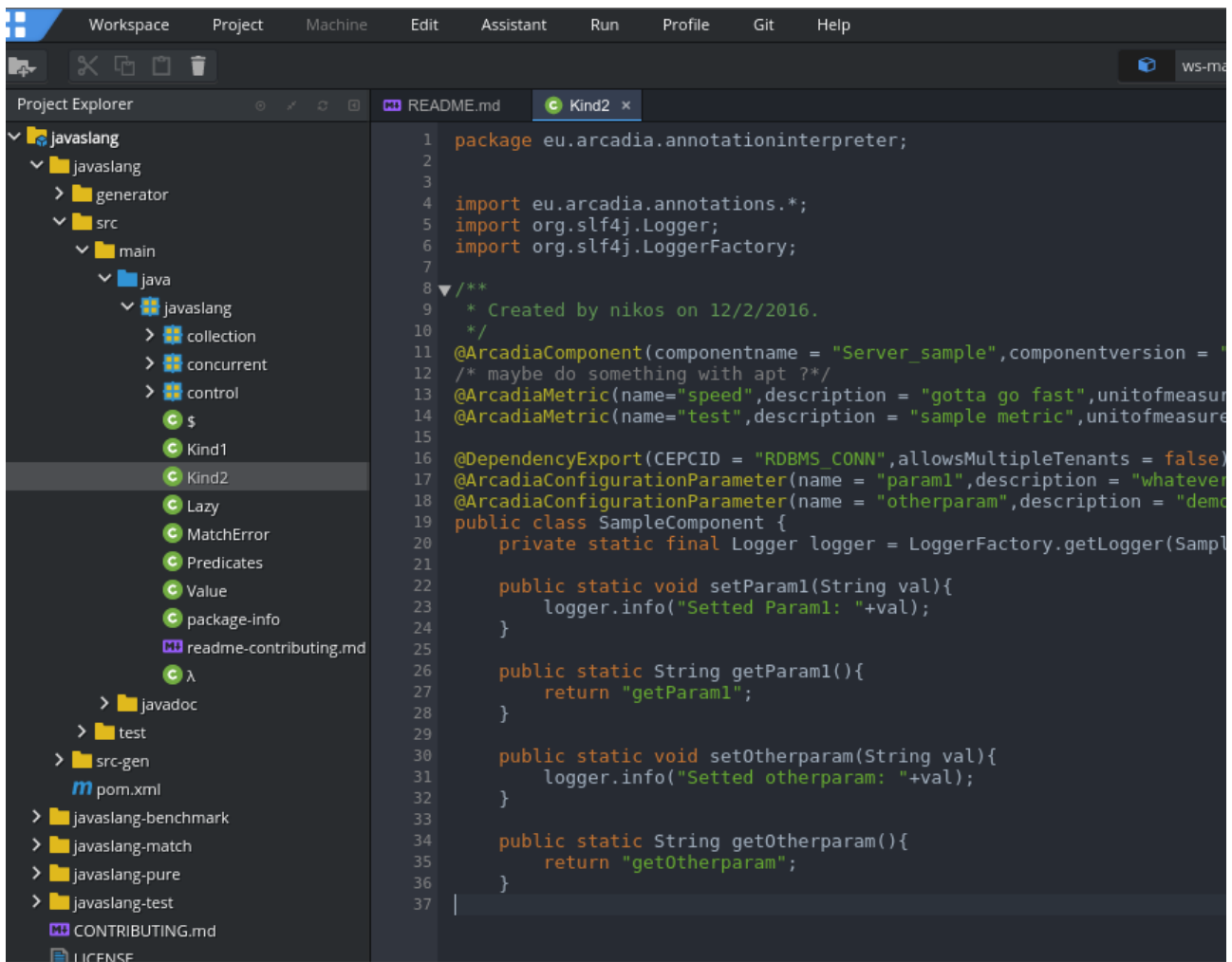


Figure 5: Development of ARCADIA component through web-based IDE

### 3.2.2 Interaction with the Smart Controller

The ARCADIA web-based IDE uses Docker for managing the projects and workspaces since this is ‘inherited’ from Eclipse Che. That means that the actual code is inside a container, and a user can access it from any browser after authenticating himself. However, the web-based IDE helps developers to validate ARCADIA annotations used in the code and to push their application to the ARCADIA platform through a bridge with the Smart Controller. To check the validity of ARCADIA annotations, the ARCADIA Smart Controller uses a custom Java annotation processor to check the correctness of the micro-service. If the developer chooses “Validate project” from the ARCADIA menu, the client-side plug-in makes an HTTP request to the Smart Controller, and the annotation processing starts for the entire project. After the validation is finished, a console window opens in the IDE editor with a report about possible errors and warnings. The project is not compiled during annotation processing. Any runtime errors or possible runtime annotation issues will be resolved or reported from the ARCADIA platform during component validation, which will be examined in the next section. Another feature of ARCADIA web-based IDE plugin is the ability to push code

automatically to the ARCADIA platform. The menu “Publish ARCADIA component” will validate, compile, package and send the entire project for component validation in the ARCADIA platform. Maven is used for validation, compilation and packaging of the project.

### 3.3 ARCADIA component validation & storage

After the submission of the component to the Smart Controller, the system performs a set of validations to check the logical validity of the provided annotations. It should be noted that the structural validity of the annotations is not checked since if the project compiles correctly, structural validity is assured. However, the logical validity refers to a set of aspects such as the uniqueness of the component name, the existence of the real method-hooks that correspond to the various getters (e.g. getMetricX), the avoidance of conflicting versions, the existence of chainable endpoints, etc. All these are a small part of the logical validation, which is performed using Bytecode introspection techniques.

The maven module that performs the actual introspection is called “annotationinterpreter.” Figure 6 depicts the exact location of the projects’ source code repository where this module exists. The source code repository is located at the following URL: <https://github.com/ubitech/arcadia-framework>

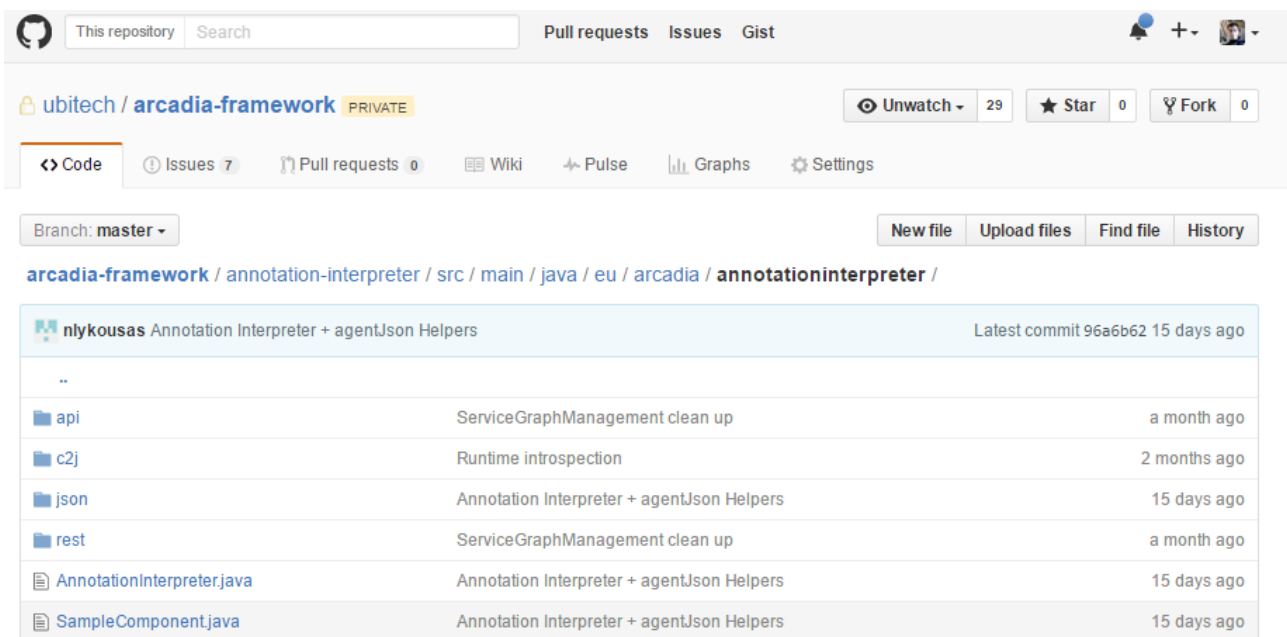


Figure 6: Maven module that performs the introspection

When the component passes the validation phase, the executable along all metadata that accompany the executable are stored in a structured format in the Smart Controller’s persistence engine. A specific parser that is embedded in the “annotationinterpreter” undertakes the task to transform the arguments of the annotations (e.g. ArcadiaComponent (name=“xxx”)) to a serialized format that adheres to the ARCADIA Context Model [7]. Figure 7 depicts part of the XSD model that represents an official ARCADIA component.

As it is depicted, each component contains a set of metadata such as its configuration parameters, the exposed metrics, and the required/exposed interfaces. The XML artifacts that are generated are indexed (using lucene [15]) and stored in the NoSQL database. The ARCADIA persistence engine is supported by three databases that are synchronized with an overlay transactional manager. These include a relational database (MySQL) which is used mainly for account management, a NoSQL database that is used for the component metadata and the monitoring output and a graph database (Neo4J [16]) that is utilized for the storage of the service graphs along with their orchestration messages.

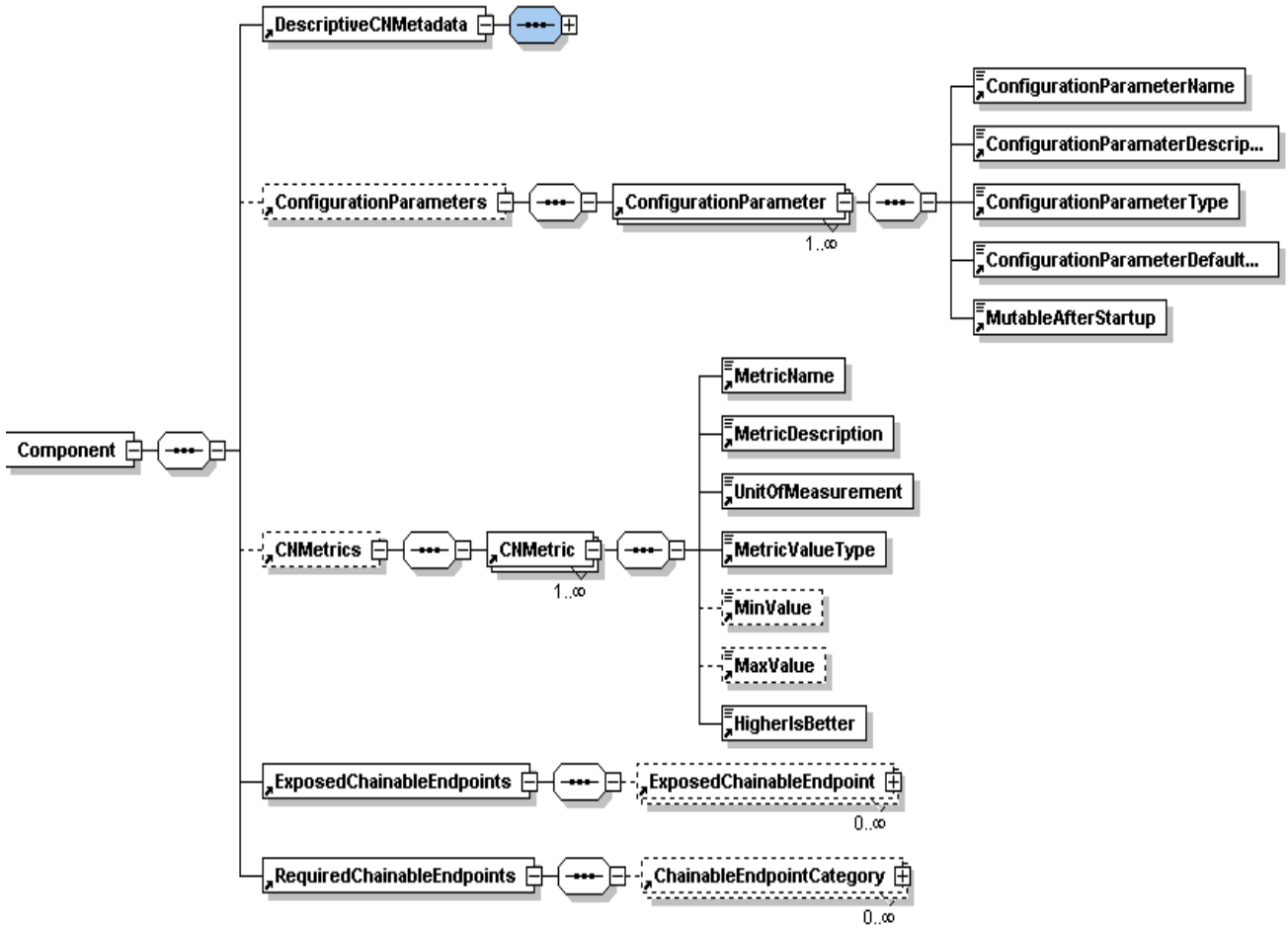


Figure 7: Serialization Model for a specific component

Furthermore, a component that is developed by a developer of a service provider can be shared with other service providers. This feature practically upgrades the ARCADIA platform to a marketplace of reusable components. Figure 8 depicts the component list-view for a specific service provider as it is presented on the ARCADIA dashboard. In the list-view, the basic metadata is presented. The details-view contains a full representation of the component including chainable exposed/required interfaces, metrics, configuration parameters, etc.

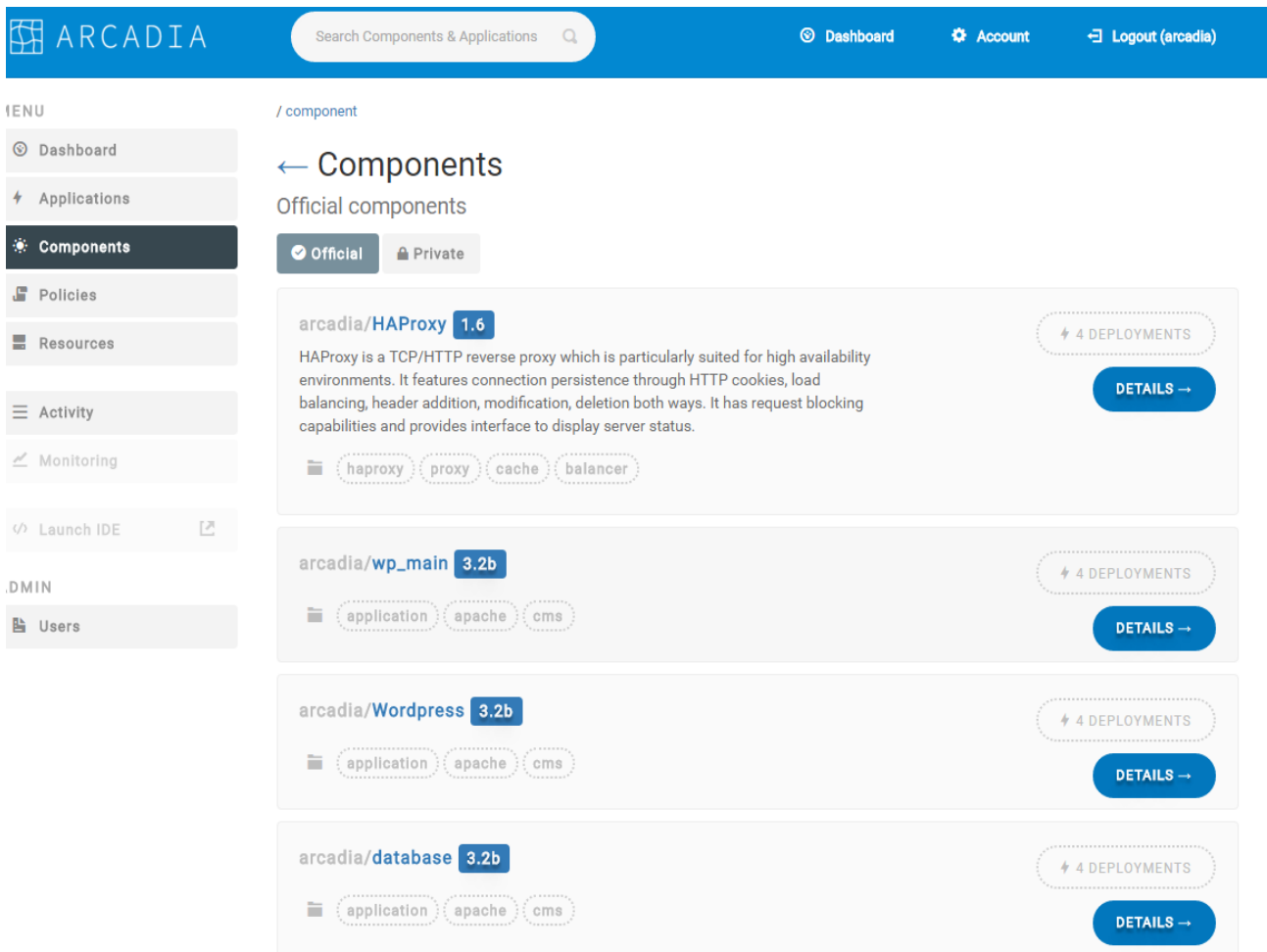


Figure 8: Components' View from the dashboard

As already mentioned above, after the development and the validation of a component, the component per se is stored in a binary format in an object store. However, this format is not suitable in order to be used directly in the IaaS resources since IaaS resources require cloud-images. The creation of the appropriate cloud-image is addressed as bundling process and will be discussed below.

### 3.4 ARCADIA component bundling

The available bundling options have been considered valid for ARCADIA includes:

- **BareMetal-bundling:** This option is the most traditional one. According to this, the microservice should be shipped as an executable, which is compatible with the kernel of the operating system that is installed in the bare-metal.
- **VM-bundling:** In this case the microservice can be shipped as a VM-Image (taking advantage of the mature standards that exist) and can be launched in different hypervisors (KVM, XEN, ESXi, etc.) that respect specific virtualization standards.

- **Container-bundling:** In this case the microservice can be shipped as a Container-Image, yet the standardization regarding a universal container is about to generate tangible results in 2016. Therefore, at this point, someone has to rely on proprietary solutions such as LXC, Docker, etc.
- **Unikernel-bundling:** This type of environment tries to combine the performance benefits of Containers (minimal boot times, small footprint, etc.) with the strong isolation guarantees of VMs. According to this paradigm, a microservice is bundled as a standard VM (i.e. it can be booted on existing hypervisors such as KVM and ESXi) however its base operating system is not a fully-fledged operating system contains only the appropriate part of the kernel (e.g. no user land space, no preloaded drivers) which is required for the execution of one service. An operational unikernel [17] has the footprint of ~5 MBs which makes it extremely lightweight regarding its execution and minimal as a microservice overhead. **Most of all, from the security perspective, a unikernel reduces the attacking surface of normal operating systems drastically.**

Although the Smart Controller of ARCADIA can be extended to support all types of execution environments, **ARCADIA adopts the Unikernel-based bundling** since it allows the utilization of traditional hypervisors without the overhead of traditional operating systems. A specific maven module undertakes the task of creating a specific unikernel per each IaaS provider since the unikernel that is produced for one provider cannot be used outside of the box to another.

After the cloud-readiness of components, which is performed using the bundling module, the next step is the composition of an HDA, which will be discussed in the section below.

## 4 Service Graph Composition

The composition of HDAs is performed using the service graph composer that will be delivered in the frame of the ARCADIA Dashboard. This component is still under heavy development, and it practically assists a service provider to compose complex applications based on primitive components.

### 4.1 Creation of Service Graphs

Service Graph composition is performed based on the compatibility of the chainable endpoints (see Figure 9). Each component may expose one or more chainable endpoints. These chainable endpoints belong to specific categories. An endpoint is represented by an identifier that is addressed as *ECEPID*, which stands for “Exposed Chainable Endpoint Identifier” while the category per se is represented by a *CEPCID*, which represents “Chainable EndPoint Category IDentifier”. As it is easily inferred, one CEPCID may correspond to multiple ECEPIDs.

For example one microservice (MySQL5.5) may expose an endpoint (e.g. FFAABB) which corresponds to the category “MySQL Connectivity” that is represented by the indicative CEPCID e.g. AA00CC while another microservice may expose a different endpoint (e.g. AABBBCC) which corresponds to the same category (i.e. AA00CC). The differentiation between CEPCIDs and CEPCIDs is crucial since the composition is relying on the principle that the chain-requestors express their request using category identifiers (i.e. CEPCID) while the chain-exporters express their advertisement using both CEPCID and ECEPID.

A specific environment that is being currently developed allows a service provider to chain two components based on their CEPCID-compatibility. When this is performed for more than two pairs, then an HDA takes the form of a directed acyclic graph. This graph is called Service Graph and upon its construction, it is persisted to the graph database (Neo4J) that was mentioned previously.

The screenshot displays the ARCADIA web application interface. At the top, there is a blue header with the ARCADIA logo, a search bar for components and applications, and navigation links for Dashboard, Account, and Logout. On the left, a vertical menu lists various application management options. The main area shows the 'Application' management page with a 'CREATE NEW' button and a service graph visualization. The graph consists of three nodes: 'database', 'wp\_main', and 'wp\_backup', connected by blue lines.

*Figure 9: Service Graph Composition*

The service graph entails a strict serialization format which is used to represent the graph. This format is in line with the ARCADIA Context Model and is depicted in Figure 10. According to this XSD model, each service graph is represented using GraphNodes and Graph Dependencies which contain additional identifiers (e.g. NodeID and SGID) that are functionally utilized by the composer.



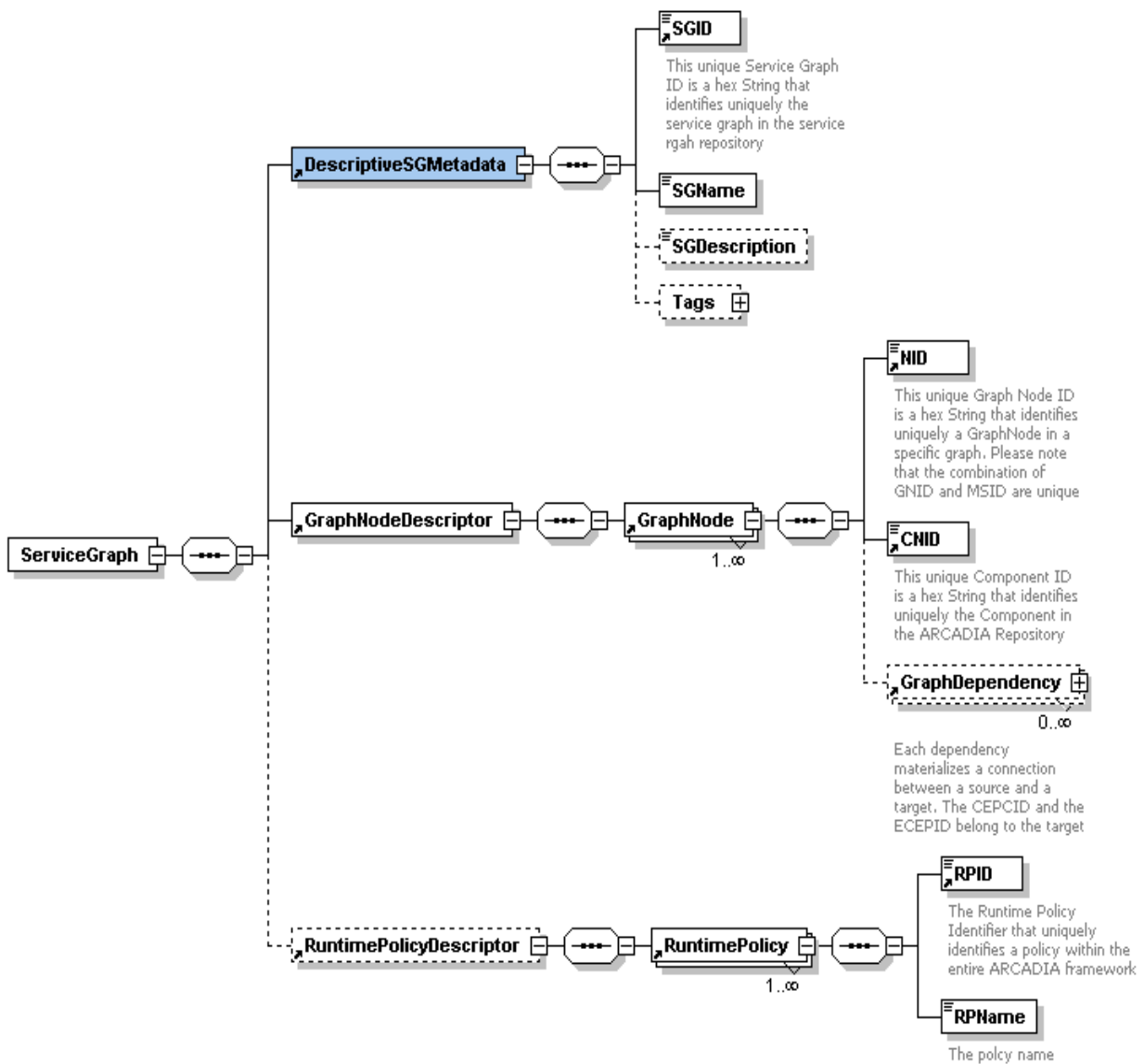


Figure 10: Serialization Model for a Service Graph

Upon definition of a service graph, the XML model that is generated is indexed and persisted. The service provider has the capability to browse among all service graphs that belong to his administrative domain. However, in an analogous manner with the components, service graphs can also be shared among other service providers. It should be noted that only service graphs that contain shareable components can be shared in order to preserve the privacy of the component developers. Figure 11 provides an overview of the application listing as it is currently ongoing developed. Through the application-list, a service provider can browse among the HDAs and even edit the service graph definitions.

Finally, as depicted in Figure , there is a set of runtime policies, which may be associated with a specific service graph. These policies define the runtime behavior of the entire service graph. Before the

instantiation of one service graph, a service provider has to select one policy, which will be used during the service graph execution. Runtime policy definition will be discussed in the next section.

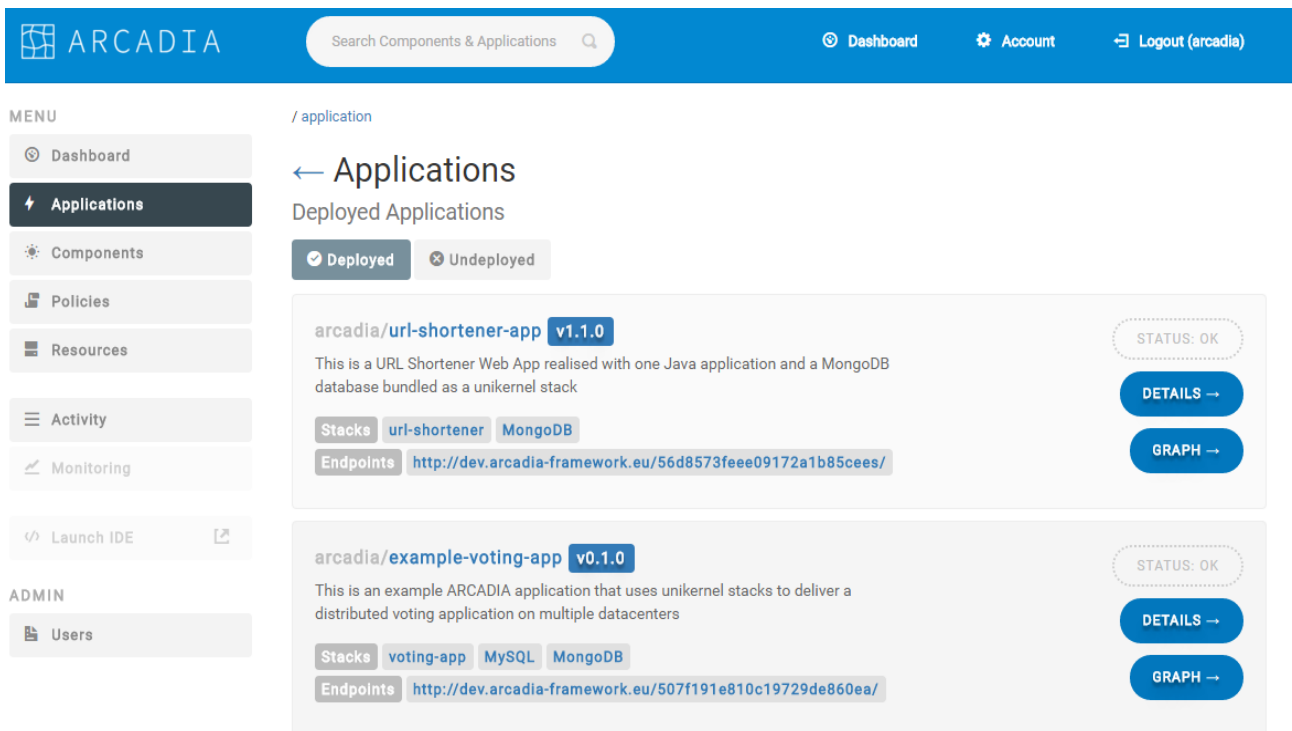


Figure 11: Overview of Service Graphs from ARCADIA Dashboard

## 4.2 Policy Management at the Service Graph Level

In general, a service graph is associated with two types of policies. One for the initial deployment and one for the execution. The initial deployment policy affects the way the various components of a service graph are placed in the different IaaS providers. Deployment is based on a set of optimization mechanisms, developed and supported by the ARCADIA Orchestrator (Smart Controller), as detailed in [1]. On the other hand, the runtime policies affect the way the entire service graph operates.

A runtime policy consists of many rules. Definition of rules per policy is supported by the Policies Editor on a per service graph basis, based on the concepts represented in the ARCADIA context model (see Figure 12). Each rule consists of the expressions part -denoting a set of conditions to be met- and the actions part -denoting actions upon the fulfillment of the conditions. Expressions may regard custom metrics of a service graph or a component/microservice (e.g. `average_response_time`, `active_users`), resources usage metrics (e.g. `memory`, `CPU`, `disk usage`, `traffic statistics`). An indicative expression is as follows: “if `microserviceX.avg_cpu_usage` is greater than 80%” and can be combined in various ways (and/or/grouping/subgrouping) with other expressions.

The potential actions of policy are classified in four categories. The first category regards the components/microservices lifecycle management (e.g. start, stop, destroy). The second category regards the management of mutable components/microservices configuration parameters (e.g. microserviceX.maxConnections set to 100). The third category regards the management of horizontal or vertical scaling functionalities per component/microservice while the last category regards IaaS management functions such as adding or removing extra VCPUs, RAM, Disk to the allocated resources to specific components/microservices.

Each rule has attached a specific salience that is being used during conflict resolution by the inference engine. A time window is also specified per rule for the examination of the provided expressions and the support of inference functionalities during this time window. When attaching a specific runtime policy to a grounded service graph, the specified set of policy rules is deployed to the engine's production memory, while the working memory agent is constantly feeding the working memory with new facts.

The screenshot shows the Arcadia web interface for editing a policy rule. The page title is "Expressions Editing" and the subtitle is "Manage your Expressions for Policy 'blogs and mysql #2 Teeeeeeeeest'". The interface includes a sidebar menu with "Policies" selected, a search bar, and a main content area. The "Policy Rule Name" is "asdf" and the "Policy Rule Priority" is set to a low value. The "Define Conditions of Rule" section contains two conditions: "wp\_main(FFFFFFE1) - disk\_usage (%) equal" and "database(FFFFFFF) - memory\_total (%) less". Below this, the "Define Actions upon the conditions fulfillment:" section shows "Components Lifecycle management" selected. The "Defined Expression Actions:" table has one row: Lifecycle Action, Wordpress (FFFFFFE2), do, start, and a Remove button.

ActionType	Component	Action	Value
Lifecycle Action	Wordpress (FFFFFFE2)	do	start

Figure 12: Creation of Runtime Policies

Before the deployment, the service provider selects one policy which will be enforced. In ARCADIA, Policy enforcement is performed using an Advanced Expert System called Drools [18]. Policies are translated into rules which are addresses as production memory while a monitoring system feeds the expert system with facts which constitute the working memory. The embedded inference engine supports reasoning and conflict resolution over the provided set of facts and rules as well as triggering of the appropriate actions.

## 5 HDA Deployment and Orchestration

The next phase in ALM is deployment. A critical aspect regarding deployment is the placement process i.e. the decision of which component is placed where among the various IaaS resources those are available. This can be done manually or based on an optimization solver, which is being developed in the frames of the platform.

### 5.1 Deployment Policies

Infrastructure consists of several physical machines able to host several execution environments (i.e. hypervisors running on physical machines hosting several virtual machines) and several physical links able to route over them several overlay channels. Hosts and links offer resources of certain capacity (capacitated resources). Indicative resources for hosts are CPU, Memory, and Storage I/O rate while links most commonly offer to capacitate Bandwidth resource. An execution environment for an HDA's software component requires a certain amount of all or a subset of resources provided by a host while a channel between software components requires all or a subset of resources offered by a link assigned for routing over it an overlay link illustrating the channel. Furthermore, assignments may require not only a certain amount of a capacitated resource but as well a certain monitored metric to be within a certain range i.e. link delay. These requirements along with others specify what it is considered a feasible assignment and form the constraints of the problem of deploying a HDA over the infrastructure.

The various constraints that have to be met prior to the assignments formulate from the mathematical point of view an optimization problem, which has to be solved in a near-real-time manner. The formulation of the problem is addressed as deployment problem and the solution to this issue is called placement or (using the web service jargon) grounding.

### 5.2 Service Graph Grounding

A service graph grounding contains the actual service graph along with the placement metadata (i.e. which IaaS provider is used per component) and the configuration metadata (i.e. which is the configuration value per each configuration parameter per each component). In an analogous manner with the component and the service graph, a service grounding is also serialized using a strict format that is defined in XSD. Figure 13 depicts part of the schema that is used in order to represent a grounded service graph.

The XML artifacts that represent a grounded service graph are also indexed and stored in the ARCADIA persistence engine. These XML artifacts are used by the policy enforcement mechanism in order to generate all the appropriate policy enforcement rules.

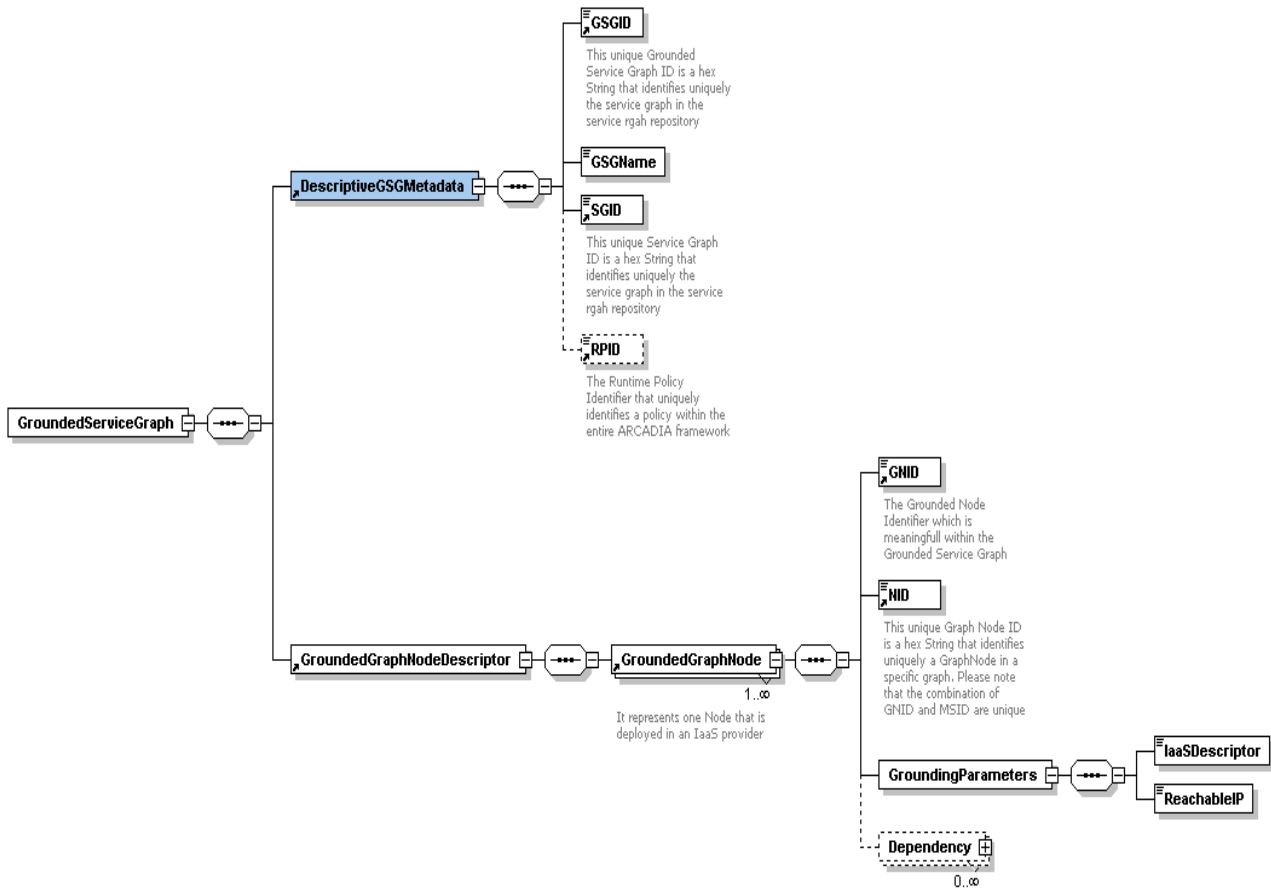


Figure 13: Serialization Model for a Grounded Service Graph

### 5.3 Runtime Orchestration

The final step of ALM is the actual deployment. The actual deployment of a component is not as straightforward as in the case of a monolithic component since there is a specific orchestration flow that has to be followed by all components in order to accomplish a harmonic deployment. For this to be achieved, each component is injected with a specific agent which is aware of the orchestration protocol. The orchestration protocol enforces a specific business-flow for the component the Execution Manager [19] that is the component that is used to orchestrate an entire service graph.

Figure 14 depicts the state space of any ARCADIA component. As it is depicted, each component, after notifying the Execution Manager (through a DEPLOYED message) that the deployment is ongoing, is blocked until all required chainable endpoints are identified. Upon resolution of the endpoints, the component enters in a STARTED mode, yet a continuous thread is listening for any potential requests for chaining by third-party components. The flow of the Execution Manager is radically different.

The Execution Manager sends a synchronous deploy message to all components that are placed in a specific resource (see Figure 15). Upon reception of this message, the components try to identify if there are specific dependencies that have to be met. If there no dependencies the components publish in a specific queue a DEPLOYED message accompanied by the identifier of the service graph which is deployed. The Execution Manager is registered to all topics that relate to graph deployments; therefore, it intercepts the messages that declare the availability of each component. In the case of existing dependencies, the components block until the Execution Manager synchronously notifies them that there is a component that is able to satisfy their dependency. It should be noted that the communication of the Execution Manager with the components is always synchronous while the communication of the components with the Execution Manager is always asynchronous i.e. through a pub/sub system.

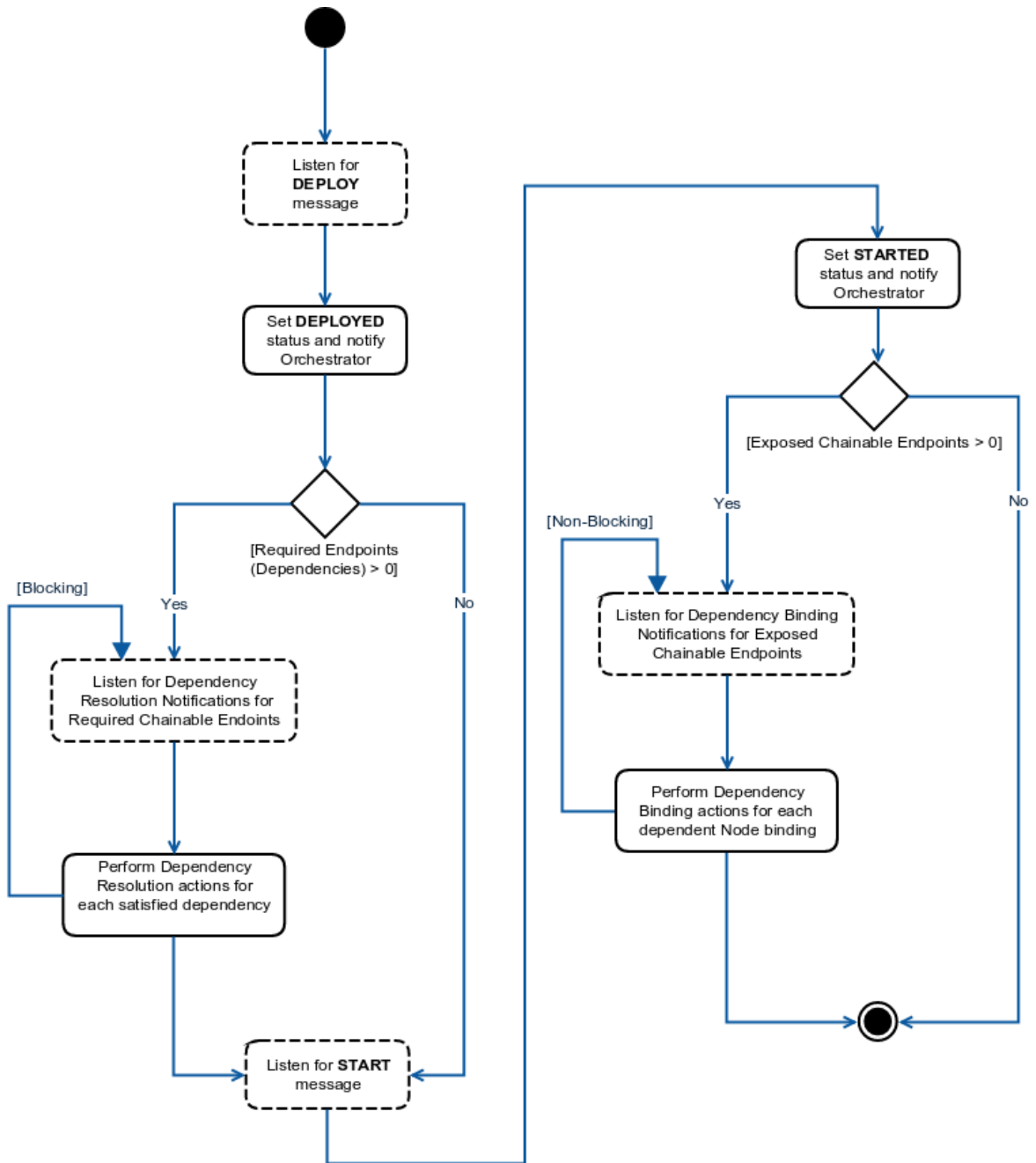


Figure 14: Component's States during deployment



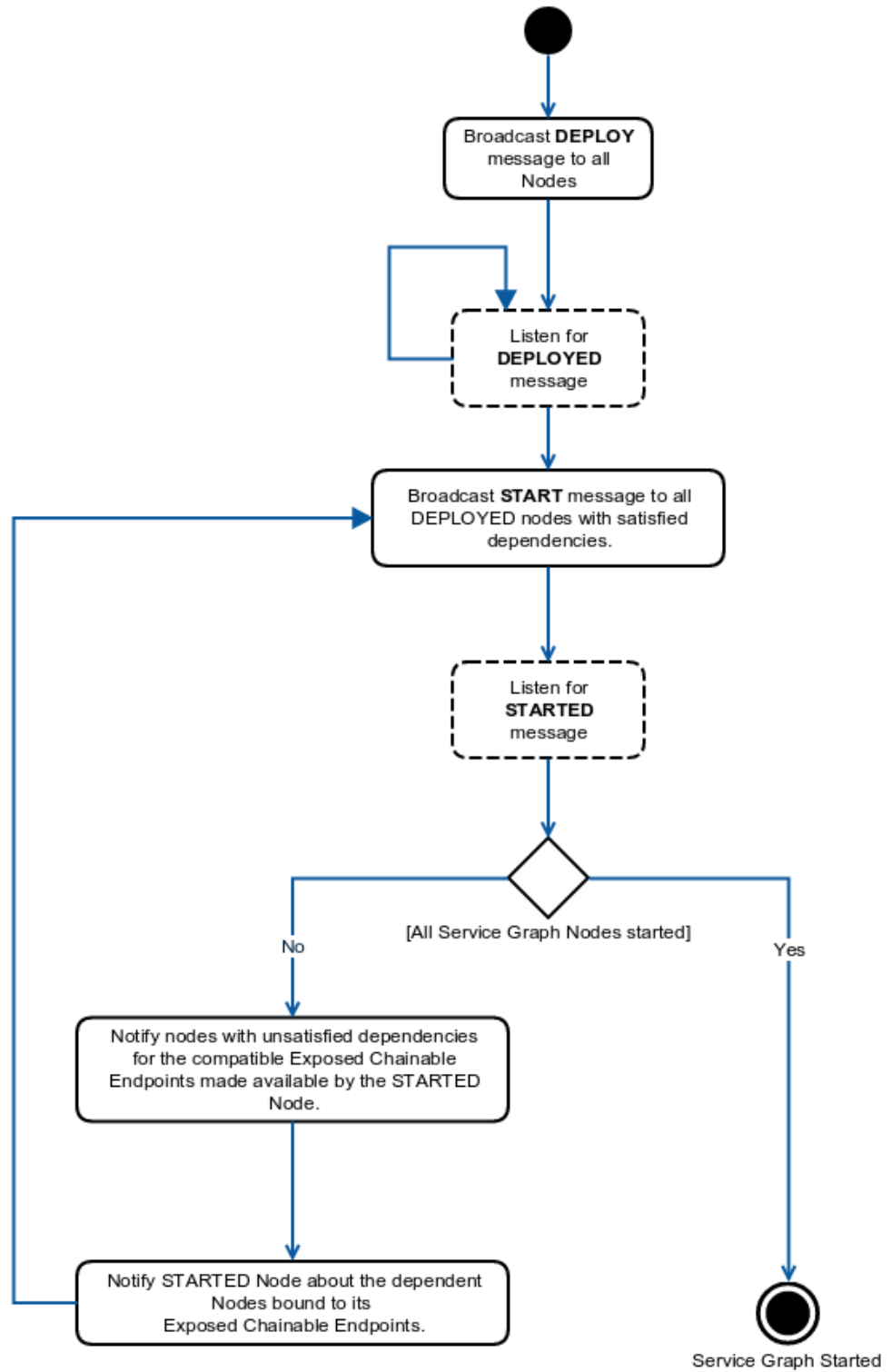


Figure 15: Orchestrator's States during deployment

## 6 Conclusions

ARCADIA aims to provide a novel reconfigurable-by-design highly-distributed applications development paradigm over programmable cloud infrastructure. A HDA, in our context, is practically a service graph consisting of components that satisfy a specific metamodel. In this deliverable, the proposed software development and operation methodology is presented, aiming to support the development and operation of such applications. Specifically, different application lifecycle management phases are taken into account including microservices development, service graph composition, deployment, and orchestration.

It should be noted that, the underlying principles/features, software components/platforms discussed in this document are going to be applied and evaluated in several use cases including: “Energy Efficiency vs. Quality of Service (QoS) trade-off Use Case”, “High Performance Survivable Communications in Distributed IoT Deployments Use Case” and “Security and Privacy Support in the FIWARE Platform Use Case”

## Acronyms

<b>ALM</b>	Application Lifecycle Management
<b>BI</b>	Binding Interface
<b>CEPCID</b>	Chainable EndPoint Category Identifier
<b>DoW</b>	Description of Work
<b>HDA</b>	Highly Distributed Application
<b>ECEPID</b>	Exposed Chainable Endpoint
<b>IaaS</b>	Infrastructure as a Service
<b>JVM</b>	Java Virtual Machine
<b>LXC</b>	Linux Container
<b>NFV</b>	Network Function Virtualisation
<b>NFVI</b>	Network Functions Virtualisation Infrastructure
<b>NodeID</b>	Node Identifier
<b>OS</b>	Operating System
<b>PM</b>	Physical Machine
<b>PoP</b>	Point of Presence
<b>QoS</b>	Quality of Service
<b>SDN</b>	Software Defined Networking
<b>SGID</b>	Service Graph Identifier
<b>VLAN</b>	Virtual Local Area Network
<b>VNF</b>	Virtual Network Function
<b>VPN</b>	Virtual Private Network
<b>WP</b>	Work Package

## References

- [1] Deliverable 2.3 - "Description of the ARCADIA Framework", ARCADIA H2020 Project, <http://www.arcadia-framework.eu/wp/documentation/deliverables/>
- [2] D. Chappell, "What is Application Lifecycle Management ?", Microsoft Corporation, 2008
- [3] Spring Boot, <http://projects.spring.io/spring-boot/>
- [4] Consul, <https://www.consul.io>
- [5] Kernel Virtual Machine, <http://www.linux-kvm.org>
- [6] OASIS Topology and Orchestration Specification for Cloud Applications, [https://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=tosca](https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca)
- [7] Deliverable 2.2 - "Definition of the ARCADIA Context Model", ARCADIA H2020 Project, <http://www.arcadia-framework.eu/wp/documentation/deliverables/>
- [8] Model-Driven Approach for design and execution of applications on multiple Clouds, <http://www.modaclouds.eu>
- [9] PaaSage: Model-based Cloud Platform Upperware, <http://www.paasage.eu>
- [10] Eclipse Che: a developer workspace server and cloud IDE, <https://eclipse.org/che/>
- [11] An open platform for distributed applications for developers and sysadmins, <https://www.docker.com>
- [12] GWT: Development toolkit for building and optimizing complex browser-based applications, <http://www.gwtproject.org>
- [13] A modern, open-source software development environment that runs in the cloud, <https://orionhub.org>
- [14] Deliverable 4.1 - "Description of the supported ARCADIA Metadata Annotations ", ARCADIA H2020 Project, <http://www.arcadia-framework.eu/wp/documentation/deliverables/>
- [15] Apache Lucene, <https://lucene.apache.org>
- [16] Neo4j: A highly scalable native graph database, <http://neo4j.com>
- [17] OSv: An Operating system designed for the cloud, <http://osv.io>
- [18] Drools: A Business Rules Management System, <http://www.drools.org>
- [19] Deliverable 3.1 - "Smart Controller Reference Implementation", ARCADIA H2020 Project, <http://www.arcadia-framework.eu/wp/documentation/deliverables/>