MORPHEMIC

Component Specification Collection & Enrichment Mechanisms

Modelling and Orchestrating heterogeneous Resources and Polymorphic applications for Holistic Execution and adaptation of Models In the Cloud

H2020-ICT-2018-2020
Leadership in Enabling and Industrial Technologies: Information and Communication Technologies

Grant Agreement Number
871643

Duration
1 January 2020 –
31 December 2022

www.morphemic.cloud

Deliverable reference
D1.2

Date
30 June 2021

Responsible partner
FORTH

Editor(s)
Kyriakos Kritikos

Reviewers
Ciro Formisano, Robert Gdowski

Distribution
Public

Availability
www.morphemic.cloud

Executive summary

Cloud applications usually have fixed configurations; this restrains the way such applications can be adapted to optimise their performance. MORPHEMIC attempts to resolve this by supplying a mechanism through which additional configurations for application components can be recommended to application developers. Such a mechanism not only requires the use of suitable ways to functionally categorise and match open-source software with application components but also to collect metadata about such software and enrich these metadata with suitable characteristics to assist in software selection. In particular, a Knowledge Base (KB) is needed, constituting the main point via which knowledge about open-source software can then be distributed and shared between the different, relevant components of the MORPHEMIC platform. As such, this deliverable focuses on the way this knowledge is modelled and is attributed to cloud applications as well as shared between MORPHEMIC components through the use of a REST API. In addition, it highlights the way this knowledge can be enriched via the use of static code analysis techniques in order to assist in the selection of software components based on their exhibited quality and security level.

The other direction of focus of this deliverable is on the way resources can be modelled and queried. Such a direction is of paramount importance as it enables to optimise the cloud application configuration at runtime through the selection of the most suitable hosting resources based on the current application context. In this way, metadata about resources need to be collected, stored and maintained while they have to be searchable within the auspices of the MORPHEMIC platform. In this respect, this deliverable explicates the way the Executionware module realisation, Proactive, has been extended in order to support the resource management task.

Finally, this deliverable draws directions for further research by: (a) listing possible ways via which the knowledge about resources could be enriched in order to support an even more precise matching of resources and thus optimisation of cloud applications; (b) pointing out ideas on how the current resource metadata collection and filtering processes can become more efficient, especially in sight of the demanding (in terms of filtering performance) component grouping functionality.

Author(s)
Maroun Koussaifi, Kyriakos Kritikos

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871643

Table of Contents

1 Introduction

1.1 Scope

Cloud applications usually have a fix form where applications components map to specific configuration classes (e.g., VM-based, container-based, etc.). This, however, limits the potential optimisation space for such applications as some configuration classes are left out while they can be promising to accelerate the performance of application components. For instance, in the case of an application that includes a compression component, while such a component could be realised in the form of a classical VM just with CPUs, its performance could be boosted through the use of a FPGA-based resources. This limitation is promised to be bypassed by MORPHEMIC, which promises to recommend new configuration classes for application components. Such a recommendation comes with the use of an Application Profiler, a composite MORPHEMIC component, which is able to crawl through open-source software repositories and find software components that functionally match application components and have additional configuration classes, which might lead to better application performance. Then, by having multiple configuration classes per application component, the MORPHEMIC platform incorporates a Reasoning module via which the best configuration class and respective configuration is selected in order to truly optimise the application at hand.

The core of software recommendation, the Application Profiler, includes various sub-components, each with a dedicated and complementary functionality. Such components, by operating over open-source software, require a proper, unique representation of software metadata. The Crawler needs to collect the heterogenous description of such software and transform it into the required representation. The Analyser needs to exploit parts of this representation in order to find the right location of the repository code to analyse while it needs to extend the representation with the discovered features from the analysis. The Classifier needs to exploit the latter features in order to derive the functional categories of the open-source software crawled.

From this analysis, it becomes apparent that all such knowledge about open-source software not only needs to be collected but also stored and maintained. Further, it needs to be queried accordingly depending on the profiling task to be supported. As such, another sub-component of the Profiler, the Knowledge Base, has been devised, which supports all tasks related to the management of the open-source software knowledge. Architecturally, both the Application Profiler and the Knowledge Base in particular have been discussed and analysed in Deliverable D3.1 [1]. Further, the Knowledge Base is planned to be updated in order to become decoupled from the Crawler, something that will be reported in forthcoming Deliverable 3.2. This deliverable plays a complementary role to the aforementioned deliverables by explicating the representation form and structure of the open-source software knowledge/metadata as well as the API that will be offered by the KB in order to query and update such metadata.

A side-effect of suggesting new configurations for components is that the Application Profiler will be able to both match and select open-source software that realises the functionality of application components. From the perspective of application design, this is a very important capability as it enables not only to close functional gaps (in terms of the intended application functionality) but also to select open-source software that fulfills non-functional requirements and preferences. Such a selection needs to be supported by the suitable description of open-source software with non-functional characteristics that are not usually part of the content of respective open-source repositories. In this respect, this deliverable investigates ways via which the open-source software knowledge can be extended via the specification of the quality and security levels exhibited by the respective software. Such ways are deeply analysed and compared and the most promising ones are suggested for implementation – the respective implementation outcome will be reported in the next version of this deliverable, D1.4.

As indicated above, enriching the modelling of application components with recommended, new configuration classes is one part of the optimisation coin. The second part is the classical one, related to finding the most suitable resources matching the application components' configuration classes in order to maximise the application performance. This classical part, while initially realised in MELODIC, is being extended in MORPHEMIC in order to support the needed multi-level application deployment comprising both the selection of the optimal configuration classes and resources for all components of a multi-cloud application. Apart from the obvious need to subsequently support the modelling of both configuration classes and resources, a structural change came to the MELODIC platform with the substitution of Cloudiator [2] with ProActive[1] (see Deliverable D4.1 [3]). Such a substitution led to the need to extend ProActive in order to support missing functionality already offered by Cloudiator. Such a functionality included the modelling and matching of resources with resource/configuration requirements coming from the application's CAMEL [4] model in order to discover a set of Node Candidates (i.e., resource offerings) per each application component (which could then

[1] https://proactive.activeeon.com/

be the subject of selection during classical application deployment reasoning). In this respect, this deliverable attempts to explicate the way resources are now modelled by ProActive with a special focus on how resource metadata are collected, stored, maintained and updated (through the use of ProActive API).

1.2 Intended Audience

This deliverable targets the following audience:

- Developers of the Application Profiler module and especially of the sub-components of this module. They benefit from the description of the presented Knowledge Base API such that they can exploit it in order to access or update the metadata of open-source software. They are also acquainted with the representation model of open-source software that has been extended in this deliverable in order to know exactly which are the main characteristics of such a software and which ones to update, when needed.
- Application owners and devops, both internal and external to the MORPHEMIC project. They can exploit the Knowledge Base API in order to obtain ideas on how to design and/or realise parts of a particular application as well as how to extend the description of application components with new configuration classes in order to give rise to polymorphic applications.
- Other developers of the MORPHEMIC platform (e.g., CP Generator developers) as well as external developers and system integrators. They can benefit from the offered ProActive API in order to retrieve resource knowledge, support the production of node candidates for applications as well as develop new deployment reasoning modules able to provide suggestions for optimising the deployment of polymorphic applications across the multi-cloud spectrum.

1.3 Document Structure

The remaining part of this document has been structured as follows:

- Chapter 2, titled as State-of-the-Art Analysis, analyses the state-of-the-art in software component and resource modelling.
- Chapter 3, titled as Resource and Component Modelling, supplies the description/representation models for both software components and cloud resources/infrastructure.
- Chapter 4, titled as Software Component Management & Enrichment, unveils the way open-source software metadata can be updated and queried through the use of a REST API supplied by the Knowledge Base.
- Chapter 5, titled as Resource Offering Management & Enrichment, analyses the ProActive Extension towards the collection, storage, querying and updating of metadata related to cloud resources.
- Chapter 6, titled as Conclusions & Future Work, concludes this document and draws directions for further work.

2 State-of-the-Art Analysis

In this chapter, we attempt to supply some background knowledge on activities reported on this deliverable also mapping to some of its main contributions. In this respect, this chapter is separated into sections reviewing work on resource modelling, software modelling as well as non-functional feature extraction.

2.1 Resource Modelling

The modelling of cloud resources has been a very active area of research with multiple approaches being proposed focusing on different description aspects and different levels of abstraction. In order to reduce the overall space of work, we focus our analysis mainly on the infrastructure level and consider mainly the functional aspect in resource modelling. The software level is covered in Section 2.2 where both the functional and non-functional aspects are touched. The non-functional aspect for the infrastructure level is not so required to be analysed as it can be potentially covered by respective service description approaches which are quite generic enough in some cases. However, this would need, of course, the usage of an agglomeration of languages and not just a single language.
Concerning the cloud resource modelling, we can categorise the relevant approaches into three classes: standardised approaches, semantic and semi-formal ones. Each from these classes is analysed in a separate sub-section of this section.

2.1.1 Standardised Approaches

Standardised approaches are those which employ either an existing standard, which might be adopted by the industry, or a recommended standard, which could be adopted by the academic community. Each from the considered approaches comes from a different organisation which justifies the existence of multiple instead of a single standard in the area.

OCCI[2] (Open Cloud Computing Interface) aims at defining an API for cloud resources on the IaaS level. This standardised interface has not been uptaken yet and very few attempts have been exercised that implement it on private cloud platforms like OpenStack[3] or cloud abstraction libraries like jclouds[4].
TOSCA [5] is an open standard recommended by OASIS for describing cloud applications and services. This standard has also a low adoption level in the market where only one cloud product, Cloudify[5], exploits it for cloud service orchestration purposes. However, the adoption of this standard in the academic world is significant.

2.1.2 Semantic Approaches

mOSAIC [6] is an OWL ontology trying to conceptualise the whole cloud domain incorporating various related concepts like cloud resources, metrics, SLAs and services. At the infrastructure level, it captures mainly well-known resources kinds but not newer ones like GPUs and FPGAs. This gap is covered by the Cloud Lightning (CL-Ontology) ontology in [7], which is actually an extension of mOSAIC. This ontology covers concepts related to various kinds of resources in heterogeneous environments with a special focus on hardware accelerated ones. Finally, a generic ontology called HPCRO has been proposed in [8] able to cover concepts related to both hardware and software resources along with their interrelationships. However, no modelling of specialised and new kinds of hardware resources is covered like hardware accelerated ones. In any case, most of the proposed ontologies seem to focus mainly on the semantic annotation of structural resource description formalisms in order to enhance their semantics and not on providing a single formalism that can be utilised individually for the complete, semantic description of cloud resources.

2.1.3 Semi-Formal approaches

Various semi-formal approaches related to resource modelling have been already covered in D1.1 deliverable [9]. To this end, we mainly cover in this section some additional, semi-formal approaches. Such approaches usually rely on a semi-formal formalism like UML or a structural-based formalism like XML Schema.
A language that supports the semi-formal description of so called blueprint templates has been proposed in [10]. These templates can specify cloud offerings at different levels of abstration as well as cover additional aspects like QoS and policy ones.
The feature meta-model [11] is a UML-based model enabling to specify in a flexible but semi-formal manner hierarchies of cloud offerings at different levels of abstraction. The flexibility is introduced via the use of specialised constructs that constrain the ways different parts of the offerings can be combined with each other. The model seems to support a complete description of the offerings by enabling to incorporate all their necessary parts. However, it needs to be accompanied with a specific ontology that semantically annotates these parts in order to enhance their semantics. Such an annotation then enables a more precise matching of the described offerings with user requirements.
Finally, in MELODIC, a specific resource description model [12] has been adopted in order to represent whole cloud infrastructures and their parts, thus being able to associate such infrastructures with the resources that they offer as well as such resources with other artifacts and elements like images and locations. However, only normal, VM-based resources were covered and not specialised like hardware accelerated ones. A Restful interface on top of the infrastructure database was also constructed enabled to query the database with specific VM characteristics in order to discover node candidates that match the resource requirements of application components. The querying and respective matching can be performed in two different ways: (a) using simple attribute requirements; (b) using OCL expressions. The simple attribute requirements are expressed in the form of <concept>.<attribute> <operator> <value> where the <attribute> (e.g., number of codes) is the attribute of interest that should belong to the specific <concept> (e.g., VM) in the resource description model. The use of OCL expressions, which enables specifying more complex requirements, relies on the approach in [2] where the matching is performed by constructing constraint satisfaction problems from the respective resource requirements and capabilities and solving these problems.

2.2 Software Component Modelling

The description of software is a vast area of research that has been also specialised for specific types of software like web services or cloud services. In the following, we will attempt to analyse both generic description models as well as those for specialised software. Emphasis will be put mainly on those approaches that focus on producing metadata describing software rather than those which attempt to represent software in different formalisms.

[2] https://occi-wg.org/
[3] https://www.openstack.org/
[4] https://jclouds.apache.org/
[5] http://cloudify.co/

2.2.1 Generic Description Models

2.2.1.1 CodeOntology [13]

This is an OWL2[6] ontology which focuses on the semantic representation of software source code. It comprises 65 classes, 86 object properties and 11 data properties. It is able to also represent partitive (part-whole) and generic (generic-specific) relations through the use of the XKOS vocabulary[7] and especially the terms *xkos:hasPart* and *xkos:isPartOf*. Further, it uses well-known design patterns like the N-ary relation[8] and the SV (Specified Values)[9]. The ontology is available here: http://doi.org/10.5281/zenodo.577939 under an CCBY 4.0 licence. A documentation of the ontology is available here: http://codeontology.org/.
Apart from the ontology, a particular RDF serialisation system has been produced able to automatically produce RDF triples conforming to this ontology from Java source code as well as bytecode. This system comprises one parser that builds an abstract syntax tree (AST) from the source code, some processors able to produce RDF triples from the AST plus link RDF resources to DBPedia[10] [14] resources as well as Apache Jena[11] for the storage and querying of the produced RDF datasets.

2.2.1.2 COMPRE [15]

The article in [15] focuses on software component matching and re-use by following an ontology-based approach as in the case of [13]. This approach relies on the Source Code Representation Ontology (SCRO) (also based on OWL), able to semantically represent source-code artifacts and especially API structures. SCRO was then extended with additional semantic component descriptions focusing on the artifacts internal structure and their interrelationships, leading to a new ontology called COMPRE (Component Representation Ontology). An interesting characteristic of COMPRE is that it is able to exploit domain ontologies, which can be used to semantically annotate, e.g., the input and output parameters of the components/artifacts. Further, COMPRE enables the description of metadata about components as well as their input and output through the use of arbitrary keywords.
This approach, apart from the proposed ontology, is able to semi-automatically produce semantic component descriptions in COMPRE which are then serialised into RDF and handled by the Apache Jena framework for storage and querying. The production of SCRO component descriptions is automatic as well as the generation of the metadata through parsing of the component source code via Apache Lucene[12]. However, the annotation of components based on COMPRE's object properties (e.g., representing component dependencies) must be done manually.

2.2.1.3 Sourcerer [16]

Sourcerer is an infrastructure for large-scale collection and analysis of source code mainly for Java. This infrastucture's architecture is data-centric and comprises multiple levels: tool, stored content, service and application. The tool level comprises tools able to perform various tasks like the crawling of source-code, the extraction of code features, code indexing and data serialisation/importing. The stored content level is a classical data-base level comprising a managed repository, the SourcererDB and the Code Index. On top of this level and respective content, various services are offered on the next level like relational querying, code search, similarity calculation. Finally, the topmost level includes application built on top of these services which comprise Sourcerer Code Search, CodeGenie and Sourcerer API Search.
Focusing on the second level, the stored content one, the managed repository is a huge project repository containing a local copy of open-source software projects which has been collected from various forges. The crawling is done periodically while there is only one version of each project maintained, the head revision one to keep the repository's size to the possible minimum. The repository follows a particular structure especially for each individual project, where both zipped and the actual content of the project is captured along with a file with very few generic metadata (like the project's name and original repository URL).
On the other hand, the SourcererDB is a relational database that stores the description of the software projects in terms of a particular relational meta-model, which is Java-specific. This meta-model focuses mainly on structural elements in the source code, such as packages, classes, and methods, which are classified as either declared or type entities. It is an extension of [17] with particular Java 1.5 features. The decision to choose and extend [17] was based on two main

[6] https://www.w3.org/TR/owl2-overview/
[7] https://ddialliance.org/Specification/RDF/XKOS
[8] https://www.w3.org/TR/swbp-n-aryRelations/
[9] https://www.w3.org/TR/swbp-specified-values/
[10] https://www.dbpedia.org/
[11] http://jena.apache.org/
[12] https://lucene.apache.org/

requirements: (a) sufficient expression level to express fine-grained and structural queries; (b) efficiency and scalability to cover the vast amount of available software projects. The different code entities captured by this meta-model are related to the project and specific file from which they have been extracted as well as to other code entities through a specific set of relations.
Finally, the Search Index allows searching over both the Managed Repository and the SourcererDB. It has been built by using the Apache Lucene information retrieval engine as well as the Apache Solr[13] interface for index creation and advanced ranking capabilities.

2.2.1.4 DOAP[14]

It is an RDF/XML vocabulary for the semantic description of (open-source) software projects as well as their associated resources. This vocabulary has been designed in order to support an internationalized description, the production of tools for the generation and consumption of its semantic descriptions, the interoperability with other web (metadata) projects like RSS[15] & FOAF[16] and its easy extension for specialised cases. It currently includes three main classes, namely Project, Version and Repository with connections between them. Each class is associated with a rich set of metadata. DOAP is associated with various kinds of tools able to process its descriptions, including validators, generators, viewers, aggregators as well as web sites that use DOAP. Please note that aggregators[17] represent a specific kind of tools able to aggregate a set of DOAP files for a specific purpose like the generation of a project registry. Due to the existence of such tools, DOAP seems to have gained some popularity with respect to other formalisms/software representations. For instance, the Apache Software Foundation (ASF) now uses DOAP in order to produce a web site that provides a proper description of all its software projects as well as different ways to view information about them. It should be remarked that ASF has produced its own extension of DOAP in order to cover some additional, required information aspects for its software projects like the Project Management Committee.

2.2.1.5 ADMS.SW[18]

It is a metadata vocabulary, produced in the context of the joinup[19] platform (software forge) of the European Commission (EC), aiming at describing free and open-source software for supporting the exploration, discovery and linkage of software on the Web. It has been built from other approaches/vocabularies including DOAP, SPDX[20], ISO 19770-2[21], ADMS[22] and Trove software map[23]. The vocabulary can be considered as rich, covering various metadata and entities about open-source software, incorporating interesting relations like *forkOf* between software projects and being linked with SKOS ontologies mapping to aspects like programming languages, topics and operating systems. Apart from the vocabulary, a spreadsheet-based tool[24] is offered which enables the modeller to supply the needed metadata for the software which are then transformed into an RDF description complying to the proposed vocabulary as well as a ADMS description validator[25].
ADMS.SW relies on the ADMS metadata vocabulary (version 1.0) which focuses on the description of interoperability assets. While ADMS.SW reached a final version of 1.0[26], ADMS has evolved and currently has 2.0[27] as its latest version. ADMS has been created by the ADMS Working Group and DG Digit (especially the SEMIC action of the ISA[2] Programme). It has been also successfully implemented not only in the joinup platform but also the Metadata Registry (MD)[28] of the EU Publications Office[29], the PoolParty Theasurus Manager 3.1.0[30] and the XRepository[31].

---

[13] https://solr.apache.org/
[14] https://github.com/ewilderj/doap/wiki
[15] https://www.rssboard.org/rss-specification
[16] http://xmlns.com/foaf/spec/
[17] https://github.com/ewilderj/doap/wiki/Aggregators
[18] https://joinup.ec.europa.eu/collection/semantic-interoperability-community-semic/solution/asset-description-metadata-schema-software/about
[19] http://joinup.ec.europa.eu/asset/adms/asset_release/adms-application-profile-joinup
[20] http://spdx.org/
[21] http://www.iso.org/iso/catalogue_detail.htm?csnumber=53670
[22] https://joinup.ec.europa.eu/collection/semantic-interoperability-community-semic/solution/asset-description-metadata-schema-adms/about
[23] https://sourceforge.net/p/easyhtml5/tracinst/Software%20Map%20and%20Trove/#what-is-trove
[24] https://joinup.ec.europa.eu/document/generate-software-description-metadata-spreadsheet-refine-rdf
[25] https://joinup.ec.europa.eu/news/just-released-admssw-valida
[26] https://joinup.ec.europa.eu/asset/adms_foss/release/release100
[27] https://joinup.ec.europa.eu/release/adms-ap-joinup-version/20
[28] http://publications.europa.eu/mdr/index.html
[29] http://publications.europa.eu/en/web/about-us/who-we-are
[30] http://www.poolparty.biz/
[31] http://www.xrepository.de/

2.2.1.6 CodeMeta[32]

CodeMeta focuses on providing a vocabulary of metadata (a kind of https://schema.org suggestion) for describing scientific software. It has been developed by accounting for the scientific software management lag problem which relates to a lack of unity or interoperability. In particular, various software projects may be managed by different forges and meta-repositories with no interoperation and thus ability to search over such projects in order, e.g., to find suitable code and extend it for research purposes. Apart from the specific vocabulary that is being proposed, various mappings between this vocabulary and other software description formalisms or representations like DOAP are suggested. By inspecting the proposed vocabulary, we can certainly deduce that it is quite rich in describing metadata for a specific software project (with a level comparable to other approaches like DOAP) but not any kind of relationship between projects.
Apart from the vocabulary and the mappings, a CodeMeta description generator is also offered as a form-based web tool[33] as well as other kinds of tools[34] (e.g., format converters and integration).

2.2.2 Description Models for Web Services

2.2.2.1 Structural Specification Languages

Various languages have been proposed for the functional specification of services where each language focuses on a different functional aspect. The languages most commonly used structurally specify the interface of a service. These languages include WSDL [18] and WADL [19], covering the description of SOAP and REST-based services, respectively.

2.2.2.2 Semantic Languages

Structural specifications are not semantic and do not capture the service behaviour so they can be the basis for low service discovery accuracy. To this end, semantic languages have been proposed to close this gap. This includes languages like OWL-S [20] and WSMO [21] which have, unfortunately, not been undertaken due to the shortage in tools that can support the semantic service specification plus the inexistent semantic expertise of service modellers. Both languages support the specification of service I/O and behaviour in terms of pre-conditions and effects. OWL-S can also describe the abstract service interface able to cover the interactions needed with the service requester.

2.2.2.3 Semi-Formal Languages

USDL[35] covers both the business and software service description. It also has a Linked-Data counterpart [22] to become more formal. Further, it covers various non-functional aspects like SLA, quality, security, cost and legal ones. [23] proposed an approach able to integrate USDL with TOSCA so as to link service selection with deployment and thus better support the cloud application lifecycle.
SoaML [24] is a UML-based language able to specify Service-Oriented Architectures (SOAs) by defining components and their inter-relationships at both the business and service levels. However, it does not deal with the internal orchestration logic of a service or any of its non-functional description aspects.

2.2.2.4 Workflow Languages

A composite service comprises other services with lower complexity that do need to be coordinated for the proper execution of this composite service. To this end, well-known workflow specification languages have been adopted in order to express the needed internal orchestration logic of a composite service like WSBPEL [13] and BPMN [14], where the second language seems to be currently uptaken. Further, semantic annotations [25] have been proposed for both languages to assist in the production of concrete service workflows from abstract ones.

2.2.2.5 Non-Functional Languages

A thorough evaluation of most existing non-functional service description languages can be found in [26]. The main conclusions that can be derived from this evaluation include the following: (a) there are certain features that discriminate one language over another, such as the formalism, the level of richness, and the complexity; (b) languages can be classified in two classes according to the lifecycle activities that they can cover. In particular, languages able to specify service quality profiles go until the service discovery activity while languages able to specify

[32] https://codemeta.github.io/
[33] https://codemeta.github.io/codemeta-generator
[34] https://codemeta.github.io/tools/
[35] https://www.w3.org/2005/Incubator/usdl/charter

SLAs potentially cover the whole lifecycle; (c) OWL-Q seems to be the most prominent language for service quality profiles but for SLAs no language seems to prevail.

2.3 Static Code Analysis for Non-Functional Feature Extraction

D3.1 [1] deliverable has already conducted an excellent analysis on what is software quality and what kind of quality checking / static code analysis tools exist. We remind here the reader that code quality can be specified as a set of quality characteristics that bear on its ability to satisfy stated or implied needs. Each characteristic in turn can map to a set of quality attributes. For instance, the testability characteristic maps to attributes like test coverage. Each attribute in turn could be related to one or more quality metrics through which it can be measured. For instance, the complexity attribute, member of the structure characteristic, could be measured by a metric called cyclomatic complexity[36].
D3.1 analysis on existing tools was not very extensive, focusing only on a subset of available tools. Further, there was no comparison of the considered tools by a set of well-specified criteria. As one of the goals of this deliverable is to find the right method/technique/tool to use in order to derive the overall quality level of a software, we have decided to make an extensive analysis of the static code analysis landspace in order to identify the right open-source tools to use in order to support this goal. Please note that we opted mainly for open-source tools as this is a core restriction and requirement that holds for the whole MORPHEMIC platform. Towards this end, we have come up with the following set of criteria that will lead our analysis:

- *Code Quality*: what are the main quality characteristics that are covered by the tool.
- *Measurability*: usually static analysis tools check best practices for code development in order to identify potential issues over the different quality characteristics. Based on such issues, they might derive a certain partial quality level for a specific quality characteristic. However, as indicated above, there is a need to also cover specific metrics able to objectively measure quality attributes. Through such metrics and the consideration/quantification of the identified issues, it could then be possible to produce a more proper and accurate quality level for a quality characteristic. In this sense, we attempt to evaluate with this criterion the ability of a tool to measure any kind of quality metric and the number of metrics that such tool can support.
- *Bug Identification*: whether the tool is able to detect any kind of error or bug in the examined software.
- *Security*: whether security issues (e.g., vulnerabilities, hot spots) can be also identified by the tool.
- *Programming Language*: Software can be written in one or more programming languages like Java or C. In this sense, it is imperative that a tool is able to analyse source code written in multiple language and not just one. Our languages of focus are mainly Java, C & C++. Java is one of the most popular languages as it can be witnessed in Github[37] and other repository management media. On the other hand, C/C++ is also popular while it is usually utilised for the development of software that relies on hardware accelerated resources like GPUs and FPGAs. Thus, with this criterion, we examine both the number of languages supported by a tool where the higher is this number the better as well as the special support towards our languages of focus. Please note that the support for different languages by a single tool could vary. In this sense, it could be possible that one needs to combine multiple tools in order to achieve the same level of support for a magnitude of programming languages.
- *Comments*: Some additional description that could be relevant for the comparison.

Based on the devised set of criteria, the next goal was to discover the tools to evaluate. For this reason, we relied on the list of tools already slightly analysed in D3.1 and extended this list by following a dual approach:

- Searching over popular search engines like Google with proper keywords like “static”, “analysis”, “code”, and “quality”.
- Checking sources of scientific articles like Google Scholar and Web of Science and applying the snowball method [27] when references to other tools were supplied.

At the end, after obtaining a huge list of respective tools, we filtered out obsolete ones to come up with a final list of 13 tools. Such tools can be characterised as full-fledged or linters. In the first case, the tools supply additional functionality apart from linting, e.g., in terms of obtaining useful insights about various aspects of the software project at hand. Finally, based on the documentation of the tool from one or more sources (e.g., web page, repository, article), we assessed the tool against the devised criteria. The comparison result is shown in the following table.

[36] https://en.wikipedia.org/wiki/Cyclomatic_complexity
[37] Github Language Stats (madnight.github.io)

Tool	Licence	Code Quality	Bugs	Measurability	Security	Progr. Languages	Comments
Sonarqube[38]	Open-Source	Reliability, Maintainability, Structure, Dynamic Behaviours, Correctness, Documentation, Testability	Yes	Many metrics per quality attribute /characteristic	Vulnerabilities, Hot spots, Remediation effort	17 including Java & C/C++	
Code Climate[39]	Dual	Testability, Maintainability	No	Very few metrics	No	12 including Java	
CodeCov[40]	Dual	Testability	No	Just for test coverage	No	25 including Java & C/C++	
Codacy[41]	Dual	Testability, Readability, Dynamic Behaviour, Documentation, Structure, Correctness	Yes	Yes	Vulnerabilities	35 languages including C, C++ & Java	
ConQAT[42]	Open-Source	Structure, Readability, Testability	Yes	Very few	Vulnerabilities	6 languages including Java & C/C++	
Semgrep[43]	Open-Source	Performance, Correctness, Maintainability, Dynamic Behaviour	Yes (bugs, errors, logic issues)	Rule violation metrics	Vulnerabilities, Hot spots	16 languages including Java & C	Most of the rules seem to focus on security rather than quality
Squale	Open-Source	Maintainability, Reliability		Yes		4 languages including Java & C/C++	
CheckStyle[44]	Open-Source	Correctness, Structure, Documentation, Readability	Yes	Yes	Vulnerabilities	Only Java	Vulnerabilities found only through a tool like FindSecBugs[45]

[38] https://www.sonarqube.org/
[39] https://codeclimate.com
[40] https://about.codecov.io/
[41] https://codacy.com
[42] https://www.cqse.eu/en/news/blog/conqat-end-of-life/
[43] https://semgrep.dev/
[44] https://checkstyle.sourceforge.io/
[45] https://find-sec-bugs.github.io/

Mega-Linter[46]	Open-Source	Depends on the linter used / Language-specific	Same as prev. cell	Same as prev. cell		26 languages including Java & C/C++	
OCLint[47]	Open-Source	Structure, Readability, Correctness	Yes	Few		C, C++, Objective-C	
QALAB[48]	Open-Source	Testability, Structure, Readability, Documentation, Dynamic Behaviour	Yes		Vulnerabilities	2 languages including Java	
CppCheck[49]	Open-Source	Correctness, Documentation, Dynamic Behaviour, Maintainability	Yes		Vulnerabilities	Only C/C++	
Clang-Tidy[50]	Open-Source	Correctness, Documentation, Dynamic Behaviour, Maintainability	Yes			Only C/C++	Able to perform OpenCL, MPI, OpenMP, LLVM checks

From the above table, various observations can be made. First of all, most of the tools offer a single, open-source licence and only three of them offer also a closed-source licence. Please note though that the free version of the tool could not be always provided as in the case of Codacy.com, which needs to inspect the respective request and especially the size of the requesting non-profit organisation before deciding whether it will hand over this version or just give a discount. Unfortunately, while the other two tools do offer an unconditional free version, their performance is quite limited with respect to the designated criteria.

By considering all the criteria considered, a clear winner can be nominated which is Sonarqube, scoring well under all criteria and especially on the code quality, security, and measurability ones. The second best seems to be Codacy which is, however, restricted with the above potential possible limitation. In the third place, we could put Semgrep with the big question of whether we could deduce appropriate metrics in order to cover multiple quality attributes. Finally, an open in terms of performance tool is Mega-Linter, which is an agglomeration of language-specific linters. In this case, this tool can have a varying performance depending on the performance of the individual linters utilised. Thus, it needs to be inspected whether there is suitable performance in most of the criteria for the 3 main programming languages of focus.

By considering each criterion alone, we nominate again Sonarqube as the best in terms of the code quality criterion with 7 quality characteristics being supported. Next comes Codacy with a support for 6 characteristics. In overall, we see an average of 3-4 characteristics as a support which is good but not optimal. Further, 4 quality characteristics seem to be supported by more or less half of the tools: Reliability, Maintainability, Testability and Correctness. Thus, the support for the remaining characteristics needs to be definitely improved. In overall, the tools need to evolve in order to support most of the quality characteristics and not just 3-4 of them in order to increase their usability and added-value.

For the bugs criterion, we can observe that most of the tools (9/13) are able to detect bugs within the code of a software project. From all these tools, we can discern Semgrep, which advertises the identification of both bugs, errors and semantic/logic issues. In our view, this is quite important knowledge towards the improvement of software to become more reliable and bug-free.

[46] https://nvuillam.github.io/mega-linter/
[47] https://oclint.org
[48] http://qalab.sourceforge.net/
[49] https://github.com/danmar/cppcheck
[50] https://clang.llvm.org/extra/clang-tidy/

Concerning the measurability criterion, from a first, quick glimpse, we could say that the overall performance is good as most of the tools (9-10/13 with a question mark on Mega-Linter) do provide some measurability support. However, from the side of the measurability level, we could only discern Sonarqube as the best, especially as it is able to compute quality metrics for all the quality characteristics that it supports. The other tools provide a quite low measurability level supporting very few and quite basic quality metrics.
As far as security is concerned, we can see that only half of the tools are able to detect some security vulnerabilities exhibited by the examined software project, where one of them is able to support such a detection through the integration of another tool (FindSecBugs). That is not a very good overall result indicating that there is great space for improvement for the static code analysis tools. From those tools that exhibit the vulnerability detection capability, two can be mainly discerned. First, Sonaqube, again a winner also for this criterion, is able to also detect hot spots as well as estimate the effort needed for remediation of the discovered security issues. Second comes Semgrep with the additional capability to also discover hot spots.
For the last criterion of the programming languages (support), Sonarqune surprisingly comes third (with 17 languages) as it is surpassed by Codacy with 35 languages and Codecov with 25. In any case, we believe that by, e.g., applying a low threshold of 10 supported languages, we can observe that almost half of the tools are able to bypass it. This is a good result, especially if we consider that such a support could be considered as satisfactory. By looking at the three focused languages, the results become even better as all of these languages are supported by most of the tools. In particular, Java and C are supported by 10 tools while C++ with 9.
We should highlight at this point, related to the previous paragraph/criterion, that there is one tool that goes the language support one step further. In particular, the Clang-Tidy tool is able to recognize specialised C dialects like OpenCL & OpenMP and thus provide support for customised checks for these dialects. This looks like a special capability of this tool that makes it amenable for exploitation for our own purposes with the main logic that it does support dialects that are utilised in the development of high-performance software. We foresee that such a capability will not be exhibited by other tools, which probably will apply generic checks over C/C++.
To conclude, we consider that Sonarqube is a tool that should be surely selected due to its performance in all of the considered criteria. We also select Clang-Tidy due to the previous rationale as well as Semgrep due to its ability to detect errors and especially logic/semantic issues. Thus, we believe that an agglomeration of these three tools would enable to fully realise our envisioned approach for computing the quality and security level of open-source software projects. In any case, we leave the field open for the use of additional tools whenever we encounter a missing gap (e.g., missing support for a quality attribute) during our implementation or a under-performance of an already selected tool (e.g., non-proper support for a particular metric or of a particular language of focus).

3 Resource & Software Modelling

In this chapter, we analyse two main contributions of MORPHEMIC and WP1 in terms of modelling different application and infrastructure components. In particular, we detail the way (cloud) resources as well as software components can be modelled through the introduction of specific conceptual models or schemas in two individual sections, respectively.

3.1 Resource Modelling

The following class diagram represents how resources (offered and deployed resources) are modelled in Executionware services. This section is divided into two parts. We first analyse how discovered/offered resources are modelled and then we present the model of the deployment process (i.e., of the deployed resources).

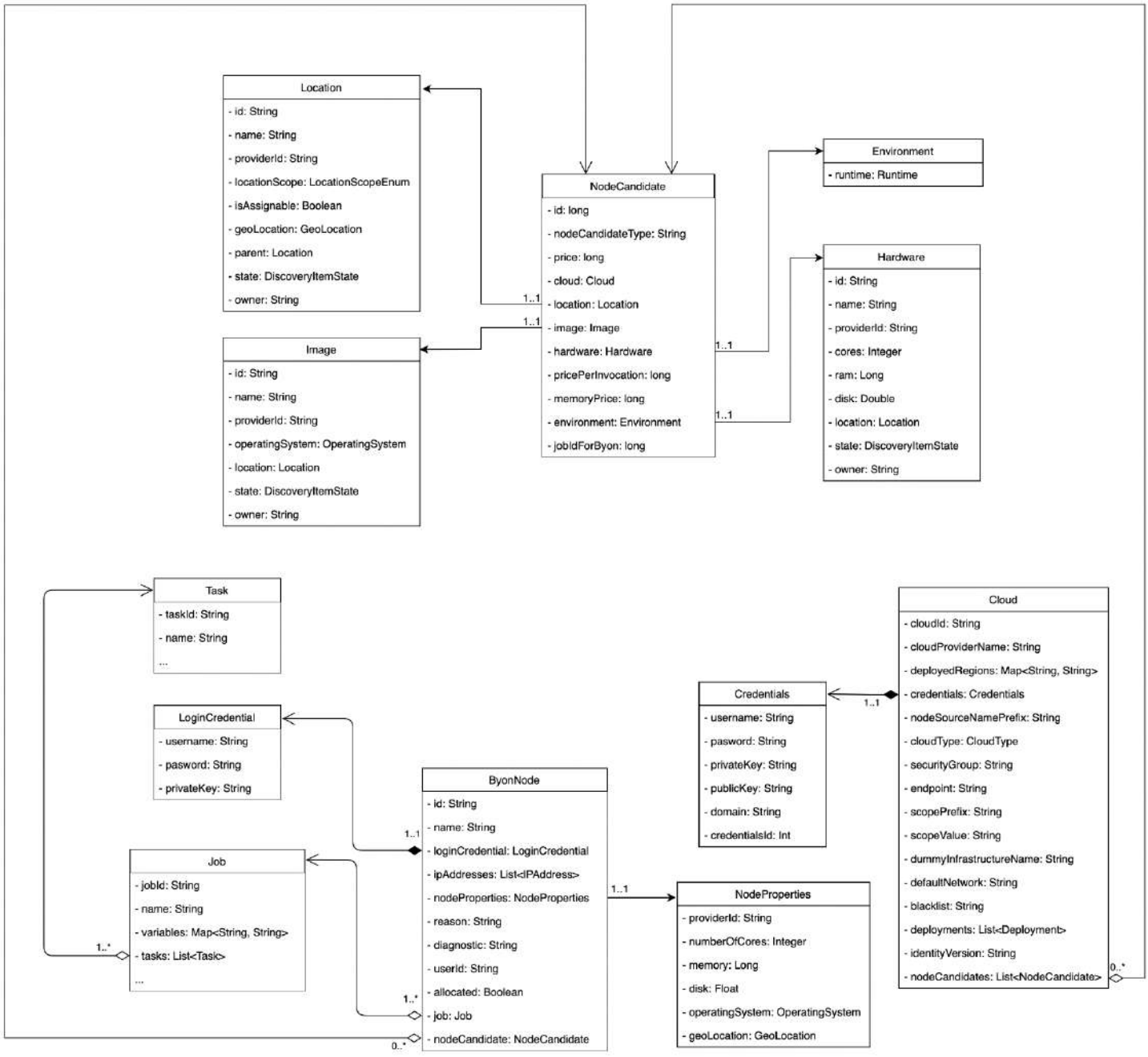

Figure 1 - Class diagram covering the resource modelling in ProActive services

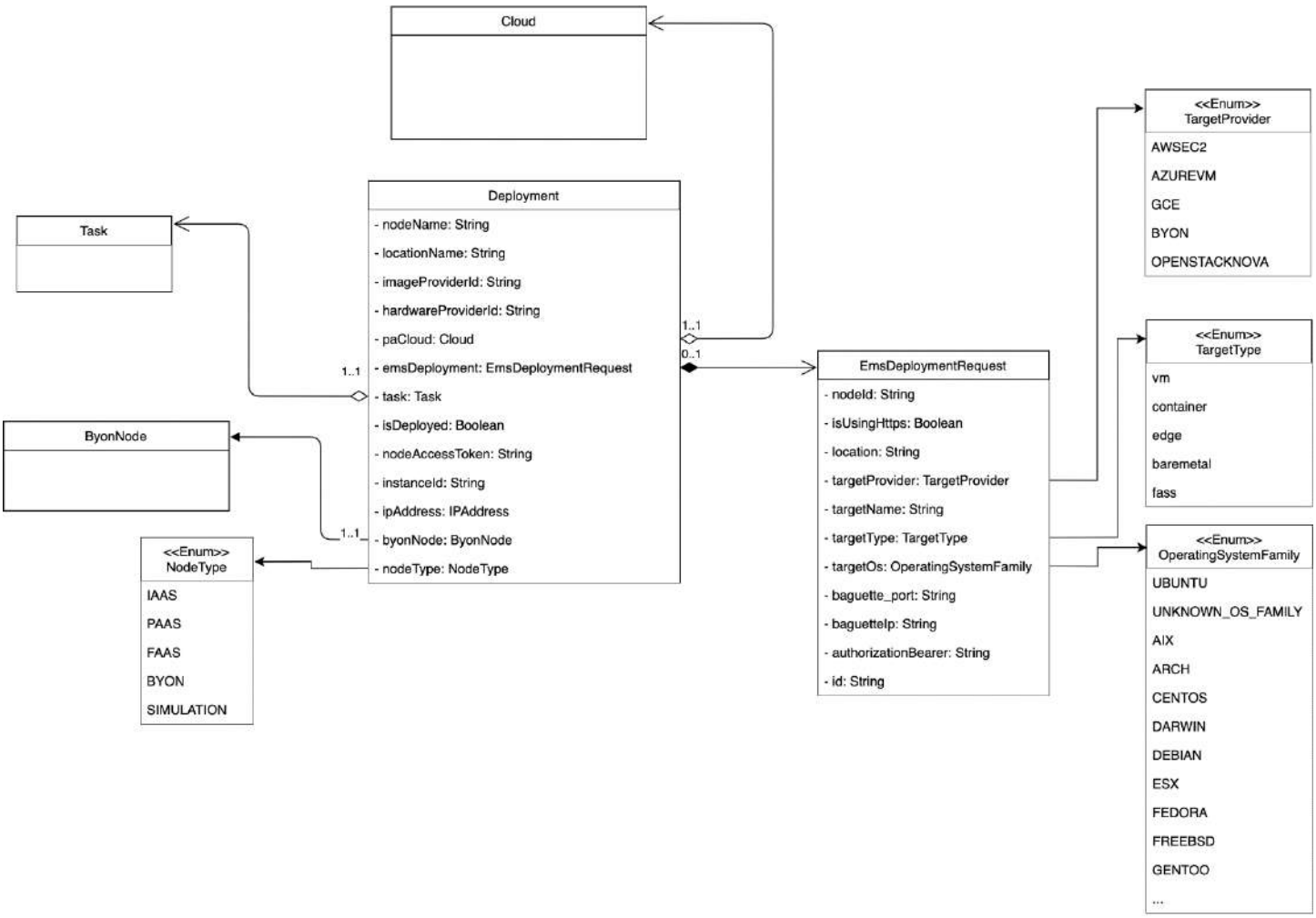

Figure 2: Class diagram covering the resource deployment in ProActive services

3.1.1 Resource modelling

The class diagram that covers the way resources are modelled in ProActive is supplied in **Błąd! Nie można odnaleźć źródła odwołania.**. As it can be seen, there is one central class called *Cloud*, which is used for modelling and managing cloud providers. This class is characterized by various attributes and properties, including:

- *cloudId*: a unique identifier for the cloud provider
- *cloudProviderName*: the name of the cloud provider
- *deployedRegions*: a map indicating where the cloud is deployed
- *credentials*: a reference to one *Credentials* object, covering credentials information that can be exploited by ProActive to perform deployments on behalf of the MORPHEMIC user in this cloud
- *nodeSourceNamePrefix*: a name prefix for the node source of this cloud
- *cloudType*: the type of the cloud. The following types are currently covered: private, public, simulation or BYON
- *securityGroup*: user's security group in this cloud
- *endpoint*: the cloud endpoint's name
- *dummyInfrastructureName*: a dummy name for the cloud infrastructure
- *defaultNetwork:* the default network for the user in the cloud
- *blacklist:* a list of the black-listed regions for which no node candidates will be collected and utilised for application deployments
- *deployments*: a list of *Deployment*s that have occurred in this cloud
- *nodeCandidates*: A list of *NodeCandidate*s, i.e., of the offerings/cloud services supplied by this cloud

The *NodeCandidate* class represents a cloud service/offering supplied by a cloud provider. Such an offering is characterised by various attributes and properties, such as:

- *id*: a unique identifier for the node candidate
- *nodeCandidateType*: the type of the node candidate (e.g., VM, container, serverless)

- *price*: the price of the node candidate per time unit (especially for VMs where time unit is per hour)
- *cloud*: a reference back to the cloud that offers this node candidate
- *location*: the actual location of the node candidate
- *image*: the actual image mapping to this node candidate
- *hardware*: a reference to the *Hardware* class covering the hardware characteristics of the offered VM
- *pricePerInvocation*: this is a pricing component for serverless nodes?
- *memoryPrice*: this is a pricing component for serverless nodes?
- *environment*: a reference to the node candidate's environment
- *jobIdForByon*: a job id for BYON nodes

The *Credentials* class covers the information of the credentials that the user has with a specific cloud provider, enabling him/her to connect to the cloud, use the respective services offered as well as connect to the VMs leased from that cloud. The attributes covered by this class are the following:

- *credentialsId*: an identifier for the credentials
- *username*: the username that the user has in the cloud
- *password*: the user's password for the respective cloud
- *publicKey*: the user's public key generated for the access to the cloud services (VMs)
- *privateKey*: the user's private key generated for the access to the cloud services (VMs)
- *domain*: an OpenStack collection of projects and users that define administrative boundaries for managing Identity entities

Apart from normal clouds that offer node candidates, it is possible that the user can add to the resource/node candidate pool individual resources that he/she might own and may reside in onsite environments (not necessarily cloud-based). In this case, the user will define a BYON node (see *ByonNode* class) which maps directly to some additional node candidates offered (e.g., some particular VMs). This is similar to the relation between *Cloud* and *NodeCandidate*, i.e., the BYON (node) offers one or more *NodeCandidate*s. The sole difference is that in a cloud there might be multiple nodes on which node candidates can be offered. Thus, a cloud is a collection of nodes established by a cloud provider while a BYON node is just one node that is offered by the user. The *ByonNode* class is characterised by various attributes and properties, including:

- *id*: an identifier for the BYON node
- *name*: the name of the BYON node
- *loginCredential*: a reference to a *LoginCredential* object, which includes information enabling to connect to the BYON node via SSH
- *ipAddresses*: a list of *IPAddress* objects representing IP addresses (public / private) of the offered BYON node
- *nodeProperties*: a reference to a *NodeProperties* object representing extra properties for the node, such as the the total number of cores, the total memory size, etc.
- *nodecandidate*: a list of the node candidates offered by the BYON node

3.1.2 Resource provisioning

During the deployment of multi-cloud applications, various resources can be created in one or more clouds or even BYON nodes. Such resources host application components and can execute tasks belonging to a certain job. These resources are obviously created from node candidates, thus inheriting the characteristics of the latter. While the information about node candidates, clouds and BYON nodes is critical for resource collection and discovery reasons, facilitating the application deployment reasoning process, there is also the need to keep track of the deployed resources originating from node candidates in order to have a clear and consistent view of the current, multi-cloud application deployment state and thus enable the proper application reprovisioning, when needed to take place. To this end, the class diagram in Figure 2 indicates the way such deployment state information is modelled while making a connection to the information already presented for resource modelling. As it can be seen, four core classes have been modelled, i.e., *Deployment*, *EMSDeploymentRequest*, *Job* and *Task*, which will be analysed in the sequel.

Deployed resources are modelled via the *Deployment* class. This class is characterized by various attributes and properties:

- *nodeName*: the name of the node to deploy
- *locationName*: the location in which the node will be deployed
- *imageProviderId*: the id of the node's image
- *hardwareProviderId*: the id of the node's hardware

- *emsDeployment*: reference to an *EMSDeploymentRequest* object, indicating the need to deploy an EMS agent in the created/deployed node
- *cloud*: the cloud on which the node will be deployed. A node will be deployed either on a cloud or a BYON node
- *task*: the current task that will be or is under execution in the node
- *isDeployed*: a Boolean variable indicating the node's deployment status
- *nodeAccessToken*: can be used to execute a task or all tasks of a workflow to specific nodes restricted by tokens. As a result, SAL will use the tokens to restrict the workflow to run exclusively on nodes containing
- *instanceId*: the node's instance identifier
- *ipAddress*: the IP address associated with the node
- *nodeType*: the node's type. It can be IAAS, PAAS, FAAS, BYON, SIMULATION
- *byonNode*: a reference to the BYON node on which the node will be deployed

As there is a need to deploy an EMS agent in deployed nodes to facilitate their monitoring and the monitoring of the components hosted by these nodes, such a need is covered by the class *EMSDeploymentRequest*. This class has the following attributes:

- *nodeId*: the id of the node on which the agent deployment will take place
- *isUsingHttps*: A Boolean indicating whether the agent will be using HTTPS protocol
- *location*: the location on which the node will be deployed
- *targetProvider*: the provider on which the deployment will take place
- *targetName*: the name of the deployment's target machine
- *targetType*: the type of the target system?
- *targetOs*: the deployment's target operating system (object of *OperatingSystemFamily* class)
- *baguetteIp*: the IP address of the baguette server
- *baguette_port*: the port on which the baguette server is listening
- authorizationBearer: the authorization bearer being utilised
- *id*: identifier of the request

As nodes can be deployed on BYON resources, the *ByonNode* class includes some additional attributes and properties which cover such a deployment, including:

- *diagnostic*: diagnostic information related to the deployment
- *allocated*: indicates whether the BYON resource is already allocated or not
- *job*: the actual job that needs to be executed on the BYON resource

A job is actually an aggregation of some tasks that need to be executed on deployed nodes. Jobs can be thus executed as a whole in BYON nodes (see *job* property above) and clouds. Each task in a job can be executed by only one deployed node. This is the reason there is a reference from *Deployment* class to the *Task* class. A job is characterized by information which includes:

- *jobId*: a unique identifier for the job
- *name*: a name for the job
- *variables*: a map from variable name to variable value
- *tasks*: the list of tasks (objects of *Task* class) to be executed on deployed nodes

Finally, a *Task* is a single unit of execution on deployed nodes that is characterized by various attributes, such as its unique identifier, name, implementation language, and script.

3.2 Open-Source Software Modelling

The current implementation of the Knowledge Base component of the Application Profiler relies on the DOAP RDF/XML vocabulary, which seems to be the most used description model for open-source software. In this respect and by considering the fact that the change of this description model would lead to a great refactoring of the Knowledge Base, it has been decided to maintain it and extend it for the own purposes of the MORPHEMIC project. In fact, as DOAP also advertises, it has been designed in order to enable and support such extensions.
Due to the fact that the DOAP description model will be used for the sharing of software metadata between the different components of the Application Profiler, we investigated which kind of information is currently missing from

DOAP in order to extend it accordingly. In this respect, respective requirements for this extension have been derived which can be summarised as follows:

- R1: Functional features from the different kinds of analysis that can be supported by the Analyser component should be modelled and associated with a software project. Such features need to be described in a flexible way in order to cover the different kinds of functional features that are expected to exist like bag of words and program (control and data flow) graphs.
- R2: Non-functional features should be also covered especially in terms of the software's level of quality and security. Again, some flexibility needs to exist as it could be possible to have different analysis methods/algorithms for the non-functional aspect.
- R3: Each software project and especially its specific releases need to be associated with one or more configuration classes (e.g., HPC, serverless).
- R4: A project's release might also be related to a non-functional (performance) model that could be derived from the Performance Module component of the MORPHEMIC platform.
- R5: The functional matches of software components of the user application could be also modelled to assist in the rapid retrieval of alternative software configurations for the application components. The rationale here is that any kind of update in the matches for an application component is immediately applied by the Matcher (component of the Application Profiler). Then this latter component, whenever there is a request from the user/devops to see some configuration suggestions, it rapidly retrieves the matches and then ranks them according to their quality and security levels.
- R6: The profile of the user application needs to be also covered spanning both the functional and non-functional aspects.

Based on the designated requirements, DOAP has been extended in various directions. Concerning R6, we have considered that both user (cloud) applications and software projects should be described in the same way. This enables to support symmetric matching as well as to have the ability to re-use application components in the context of new (cloud) application that might need to be developed in the future by an organisation that exploits the MORPHEMIC platform. However, to fully satisfy this requirement, we have included a new object property called *comprises* with the (software) *Project* as both its domain and range. In this way, we enable the composition of software so as to be able to model complex software components and thus applications.

In order to cover requirements R1 and R2, we have moved towards the direction of adopting the modelling innovation introduced also in the CAMEL language in terms of enabling the flexible description of feature models as hierarchies of features which have their own attributes/properties. In this respect, we have introduced the *Feature* concept with two main sub-concepts, the *FunctionalFeature* and the *NonFunctionalFeature*. This feature concept relates to itself via an object property called sub-features (please see the naming as we have attempted to retain the naming pattern in DOAP using '-' between the words of a property or attribute name). It also relates to its attributes via an object property towards a new concept called *Attribute*. The latter can be considered a kind of a key-pair representation. As such, it contains a *name* (an overall property that can characterise any kind/sub-concept of *Resource*, the root concept in the ontology world), a *value* attribute (with a *Literal* data type property) to capture simple attribute values and an *object-value* object property to capture composite values in the form of concept instances. Finally, we highlight that we can distinguish between the different kinds of functional and non-functional features by introducing specialised sub-concepts of the *FunctionalFeature* and *NonFunctionalFeature* concepts, respectively. For example, for the *FunctionalFeature*, we have introduced the sub-concepts of *BagOfWords* and *ProgramGraph*. On the other hand, we have introduced the sub-concept *RankTree* in order to represent a ranking tree (for ranking a software based on quality and security attributes metrics in multiple levels of abstraction) for the *NonFunctionalFeature* concept.

This modelling extension can be considered as quite flexible as it can cover various kind of functional and non-functional features. For example, in the case of a bag of words, we could have one root feature called "Bag of Words", which could have as sub-features all the words of the bag that could have been discovered by a related (static source code) analysis technique where each of these sub-features could be also characterised by one or multiple Attributes (i.e., key-value pairs). A similar representation could be followed for the representation of a *RankTree*. On the other hand, if we had the case of a Program Graph, we could simulate its representation through the use of a two-level hierarchy of features, where the root feature represents the whole graph while the leaf features represent the graph nodes. Then, the links between the different leaf features could be represented via Attributes. Please consider that in this latter case, the name of the link could follow a pattern that unveils its type (i.e., whether it is control or data (flow) link) while the (object) value would map to the target leaf feature (program graph node).

It must be highlighted that the increased modelling flexibility in feature representation also enables to bypass any kind of feature formalism restriction. For instance, by following a particular feature representation formalism, there is a lock-in to that formalism. On the other hand, the use of the current modelling can enable the use of simple

transformers which can allow its mapping to any kind of feature formalism that could be utilised by a functional (software) classification technique.

The satisfaction of the first two requirements, R1 and R2, closes via the association of a *Version* of a software project to its *Feature*s in order to properly cover the project's software evolution. In this respect, evolution scenarios could be captured where, e.g., the quality of a software project is increased from the current to the next release as well-known software development practices have been followed towards its (code) refactoring.

Requirement R3 was easily implemented through the introduction of the *Configuration* concept. This concept represents a configuration of a software project with particular sub-concepts to capture specific configurations like *HPC*, *FPGA* and *Serverless*. Similarly to the reasoning of the previous paragraph, the Version of a software project is associated to its Configuration as different project releases might adopt different configurations. For example, initial releases might follow a simple VM-based configuration while the next ones could revert to FPGA-based configurations due to the refactoring/evolution shift towards performance optimisation.

By considering the previous point, it has been decided that the non-functional model of a software project is also associated to its *Version* (Requirement R4). In order not to complicate the suggested extension, the decision to utilise a simple literal (named as *nonfunctional-model*) to represent this non-functional model has been decided. This gives some flexibility in the sense that the model could be described via a mathematical expression that could be specified in any kind of mathematical language. For instance, it could be possible to use the same modelling approach as in CAMEL in terms of the specification of composite metric formulas and utility functions. That is to rely on MathParser[51] language and use String-based expressions in that language with variable names mapping to names of non-functional attributes and metrics (e.g., specified in CAMEL).

The final extension of DOAP was related to Requirement R5 satisfaction where it was decided to introduce a simple object property named as similar-with in order to associate a software project to all the projects with which it is similar (in terms of functionality).

The extended DOAP description model/ontology is shown in the following figure where with white colour we denote the original content of this model and with grey & purple colour the newly introduced one. Please note that we have omitted most of the internal attributes of the original concepts of this model as well as particular sub-concepts of concepts like *Repository* due to economy of graphical space reasons. We should also remark that the *name* attribute is attributed to any kind of *Resource* and thus also to the *Attribute* and *Feature* concepts.

[51] http://mathparser.org/

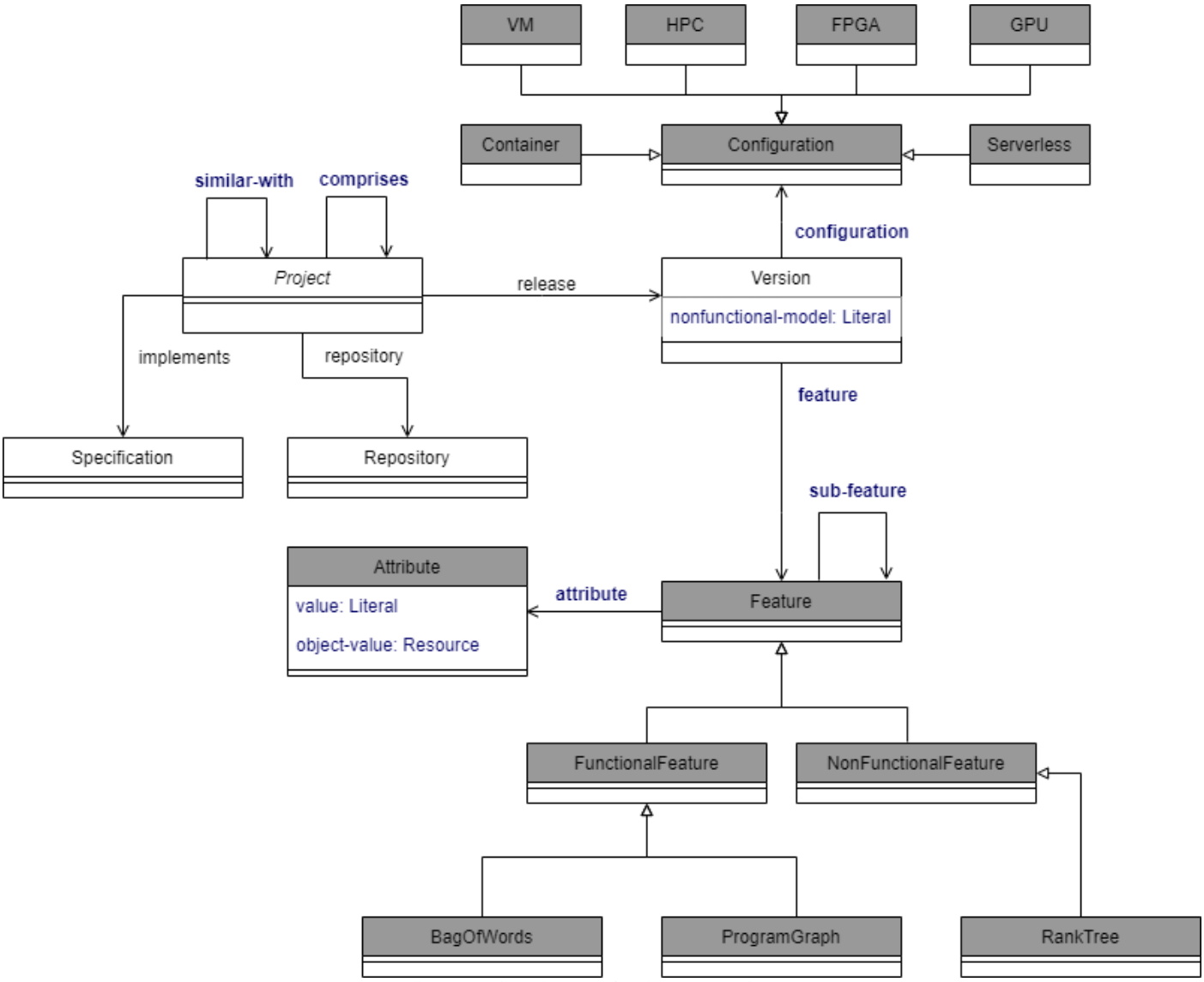

Figure 3 - The UML class diagram for the suggested DOAP extension

4 Software Component Management & Enrichment

Metadata management comes with different flavours, including their collection, storage, querying, updating and curation. Many of these tasks could be automated through the use of the appropriate algorithm, system or software. However, curation is something that is usually done manually with some computer support in particular cases. In this respect, this latter task is considered as out of focus for the analysis of the management of metadata of both software (components) and resources.

In the following, we separate this chapter into two sections. The first is dedicated to explicating how the software metadata will be managed by the MORPHEMIC platform. The second explicates how the already collected metadata could be enhanced through the production of further knowledge in the form of an overall rank of the related software based on both the quality and security aspects.

4.1 Software Component Metadata Management

As D3.1 deliverable already explained how software metadata are collected from forges and meta-forges as well as the overall architecture of both the Crawler and Knowledge Base components of the Application Profiler module or super-component, the focus of this section would be on the remaining management tasks of querying and updating the software metadata. Both of these tasks are to be supported through the incorporation of a RESTful API over the Knowledge Base component which will allow the rest of the components of the Application Profiler to attain the metadata as well as to update them in a well-defined and structured manner.

Concerning the querying part, we believe that two types of queries could be supported: (a) keyword-based queries which could be even enhanced through the use of logical operators (e.g., AND or NOT); (b) SPARQL queries that enable to query in a SQL-like way the content of the Knowledge Base in terms of the derived and stored (extended) DOAP files (for all the software projects collected). The first query form would be suitable for users with no knowledge about SPARQL who need very quickly to obtain a query result with moderate precision. On the other hand, the second query affects users with good knowledge of SPARQL who need to pose queries that produce very accurate results. Thus, as it can be seen, there is a trade-off between expertise requirements on users and query result accuracy that is targeted to be addressed through the use of two instead of a single kind of query.
Based on the above analysis, the RESTful API will supply two corresponding query methods for the two types of queries to be supported:

- softwareSearch(String) → Set<String>: a keyword-based search where the query is conducted in the form of a String while the result is a list of Strings where each String maps to the URL of the software package that is returned. See next paragraph on how to obtain additional information from this URI.
- sparqlSearch(String) → Array: a query using a SPARQL String that should return an array of results where each row will contain respective values for each variable in the query. It should be highlighted that the result of a query could be anything within the knowledge space of the extended DOAP. So, this will not be restricted to just a URI of a software package but can be parts of a software package, versions, repositories or features.

While queries can be useful in order to retrieve results that satisfy the needs of a user, the produced result might be limiting depending on the type of query posed. For instance, in case of a keyword-based query, a list of URIs is given for software projects. However, a URI by itself is not significant information. In the case of a SPARQL query, more information might be returned, which could suit the information requirements of the user in most of the cases. Thus, in order to fully cover the information needs for the first query type but as well as to enable the retrieval of whole entities from the DOAP Knowledge Base, there is a need for an (entity) access method in the RESTful API to be offered. Such a method could have the following signature:

- retrieveEntity(String, boolean) → String: here the URI of a specific instance of a Resource is specified and as a result the user/requester obtains a JSON or XML-based representation of that instance/entity according to the extended DOAP model. For instance, in case that a Project instance is required to be retrieved, then a JSON/XML String will be returned covering all the information for this project including the information about its repository and versions. On the other hand, in case that a Version instance is required to be accessed, then a JSON/XML String will be returned with information only about this software version and its included corresponding features and configurations. The second input parameter is optional and has a default value of false. It indicates whether the respective representation will include sub-entities and related entities apart from the entity to be returned. For instance, if this parameter is false, then in case we need to retrieve information about a specific software project, we will obtain only the core information about this project concerning only its main attributes/data-type properties. Otherwise, if it true, we will obtain everything about this project, including its repository and versions.

The updating of software metadata is dependent of the requesting entity, i.e., component of the MORPHEMIC platform and especially the Application Profiler module. However, please note that other components might also exploit the updating functionality, such as the Performance Module. In this respect, the following updating methods have been designed:

- updateVersion(String, String) → Boolean: This is the update method to be called by the Analyser (and thus the different analysis algorithms that might be included in that component). This method needs to be called for a specific version of the software (mainly the latest release) (first input parameter in form of a URI) and should have as a second input parameter the full JSON/XML representation of the root feature that has been produced by the respective analysis method. By supposing that each analysis method will enforce a particular naming for the root features that it produces, we envisage that two cases can occur: (a) a new root feature has been produced and needs to be stored in the KnowledgeBase; (b) an existing root feature has been updated. In the first case, a new and unique name for the root feature has to be supplied, while, in the second case, the name should be the same as the one of an existing root feature. If the requested update is successful, a true value is returned. Otherwise, a false value.
- updateSWCats(String, Set<String>) → Boolean: this update method should be called by the Classifier whenever it is able to produce a set of categories (see second parameter) for a particular software project. The software project is identified by its URI (first input parameter as String). Please note that the updating is performed in such a way that the previous categories of the software are removed and thus replaced by the new categories. The return value depends on the outcome of the update request (i.e., whether it was successful or not).

- updateNFModel(String, String) → Boolean: this update method should be called by the Performance Module whenever it needs to create or update a performance model (see second parameter, a mathematical expression in a String form) for a specific software project version/release (first input parameter, URI of the project version in form of a String).
- updateSW(String,String) → Boolean: this enables to fully update a specific software project (first input parameter as a URI in String form) based on its supplied representation (second input parameter – full JSON/XML representation of the project with all related entities included). Such an update could be performed by various components. For instance, the Profile Maintainer might apply a change that comes from the application's CAMEL model. As another example, the Crawler might discover a new version of a software package in its current crawling cycle that needs to be reflected in the description of this software package in extended DOAP.
- updateSPARQL(String) → Boolean: this is an advanced update method for experienced users who might have good knowledge of SPARQL and attempt to perform the updates through SPARQL statements. As such, the SPARQL statement is given as input while the update outcome is given as output.

The aforementioned, envisioned API supplies an added-value to the Knowledge Base transforming it into a full-fledge semantic software metadata repository that not only supports the software metadata management but also enables to perform smart queries in order to unveil further knowledge that could be useful for a devops, whether he/she seeks to support and improve the MORPHEMIC platform or an application that is managed through this platform.

4.2 Software Component Metadata Enrichment

By assuming that we will be able to produce an agglomeration of tools that do provide support for all possible quality characteristics (see Section 2.3 for a thorough analysis of the static code analysis tools and a suggestion for a tool agglomeration), we sketch here the formula for producing the overall rank for an open-source software. We make the assumption that a rank-based tree is formulated where at the top we have the overall rank, this is then split into quality and security partial ranks, next the quality rank is split into ranks per each quality characteristic and finally there can be multiple metrics per quality characteristic and the security partial rank at the leaf level. The structure of the rank tree is depicted in the following figure.

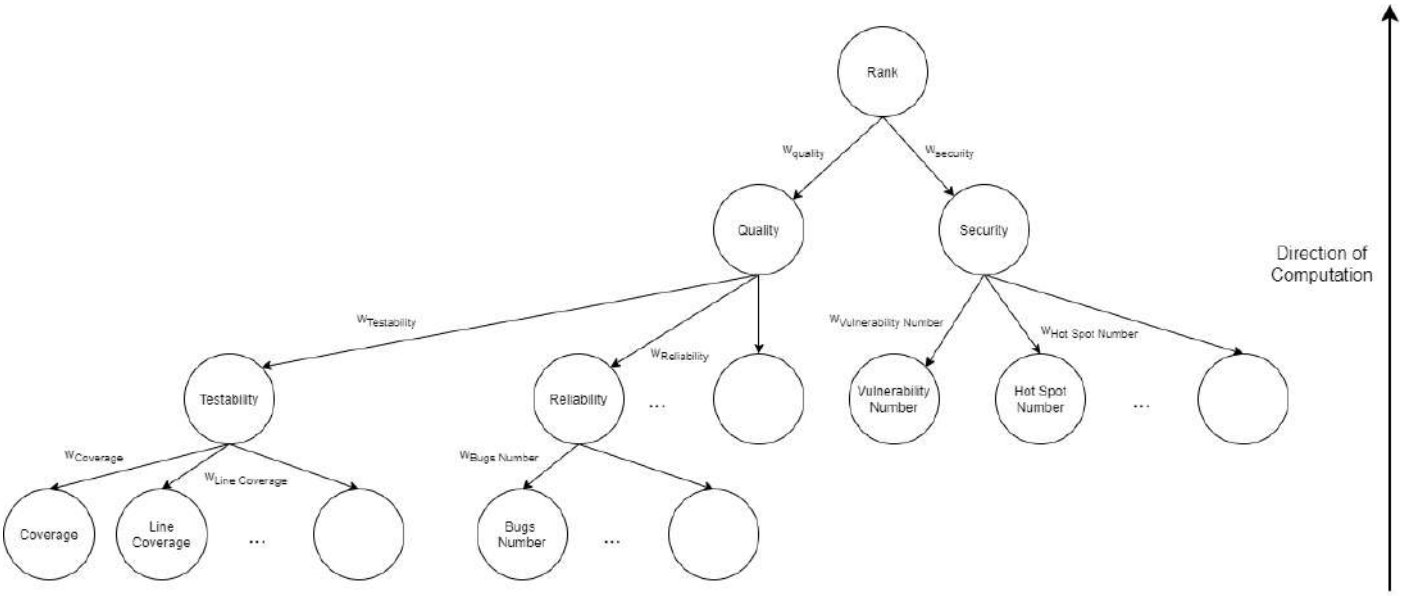

Figure 4 - The rank tree for the quality and security levels of open-source software

The rank tree represents the way the rank computation will be performed in a bottom-up manner from the leaves of the tree until its root. Each node in the tree is associated with a specific weight indicating its relative importance with respect to its sibling nodes (situated at the same tree level with the same parent node). The weights can be calculated by using the Analytic Hierarchy Process [28].
Based on the above analysis, the rank of an open-source software for a particular node can be computed depending on the node type (leaf or not):

- Leaf nodes have their rank computed directly by considering the respective, associated metric. Due to the fact that we need to produce rankings in the unit interval [0.0,1.0], it can be possible that when the corresponding metric maps to a different range, a normalisation is performed by considering a metric-specific normalisation

formula. Thus, the rank of the node - *rank(node)* - could be computed as follows: (a) either as: *metric(n,sw)* or (b) as: *normalise(n,sw)* where *n* is the name of the metric, *sw* is the open-source software at hand, *metric* is a function that computes the metric value for metric *n* and software *sw* and *normalise* is a function that computes the normalised value for metric n and software sw. The complete formula is given as follows:

  - $rank(node) = \begin{cases} metric(n, sw), n \text{ range is } [0.0,1.0] \\ normalise(n, sw), \text{otherwise} \end{cases}$

- Non-leaf nodes are computed from the weighted sum of the ranks of their children nodes. Formally, this can be expressed as follows:
  - $rank(node) = \sum_i w_i \cdot rank(node_i)$ where $node_i$ is a child of node $node$.

As the normalization formula is metric specific, all the metric normalization formulas will be presented in the next version of this deliverable, D1.4. This is also due to the fact that there is a need for a deep investigation which metrics are offered per tool that has been selected, to which quality characteristic they map and which could be good normalisation formulas/functions to normalise the values of these metrics.

Please note that most of the metrics usually compute some non-negative quantity. So, if we are interested in lower values of this quantity (imagine that, e.g., we compute the number of duplicated blocks of lines), i.e., the lower are the values, the better, then a generic normalisation function that could be utilised is: $normalise(n, sw) = \frac{1}{1+metric(n,sw)}$. This is due to the fact that when the value of the metric is 0, then we have the best utility/rank while when this value increases, then the utility/rank drops sharply from 0.5 towards 0. On the other hand, if we are interested in higher values of the metric (imagine that, e.g., we compute the number of unit tests), i.e., the higher are the values, the better is the utility/rank, then a generic normalisation function that could be utilised is: $normalise(n, sw) = \frac{1}{1+1/(metric(n,sw))}$. In this case, when the value of the metric is 0, then we have the worst utility/rank while when this value increases, then the utility/rank increases and tends to become 1.

The usage of the aforementioned, suggested utility functions is quite probable and might cover various cases of metrics. However, we will examine whether more specialised normalisation functions are needed, especially in case where we need to control how sharply is the increase or decrease of the utility/rank when the value of a metric increases. One potential solution would be to multiply the metric value with a factor/constant. For instance, in the case of the negative direction of metric preference values, the normalisation function could become $normalise(n, sw) = \frac{1}{1+a \cdot metric(n,sw)}$ where a is constant and different that 1. When a is greater than 1, then we move faster to lower utility values while when it is lower than 1, we move slower to such values. Another solution would be to utilise the negative power of e, i.e., $normalise(n, sw) = e^{-metric(n,sw)}$. In this case, the utility moves more sharply towards zero when the metric value increases. While we can also control this by using a multiplication of the metric value with a constant.

5 Resource Offering Management

To enable the MORPHEMIC platform to exploit the available resources and nodes as well as integrate with the Scheduler and Resource Manager (RM), Activeeon has developed various endpoints, found in the Scheduling Abstraction Layer (SAL), that handle this integration. Such integration endpoints enable both to add new cloud providers to the MORPHEMIC platform as well as to support the required resource discovery functionality, which is highly needed for application deployment reasoning purposes. In the sequel, we provide an architectural overview of SAL placement in MORPHEMIC and SAL components, we then provide details about the two main integration endpoints and finally we analyse the processes incorporated within these two endpoints.

5.1 Architectural Overview

Before presenting the endpoints, we provide an overview of the overall MORPHEMIC architecture in the context of resource offering management as well as briefly analyse the internal architecture of SAL module/component. The overall MORPHEMIC architecture overview can be seen in Figure 5. As it can be seen, SAL lies in between the MORPHEMIC Upperware and Executionware, playing the role of a bridge between them, hiding implementation details and peculiarities of both modules and thus enabling their easy evolution or even substitution, when the respective need arises.

Figure 5 - The overall MORPHEMIC Architecture with the SAL bridge

The internal architecture of SAL can be seen in Figure 6. SAL comprises six main components:

- *ProActive Scheduler Gateway*: it is responsible for handing the application deployment
- *ProActive Resource Management API (RM API)*: it mainly handles infrastructure deployment
- *DB Constructor*: enables to build the DB model of SAL
- *Services*: offers necessary services that handle various functionalities, such as database management, connection and user validation
- *API*: it offers the public endpoints of SAL
- *DB*: the SAL's database where information related to resource modelling and provisioning is retained and maintained

For more detailed information about SAL's architecture, please refer to the deliverable D5.3 [29] and to the SAL's GitLab repository[52].

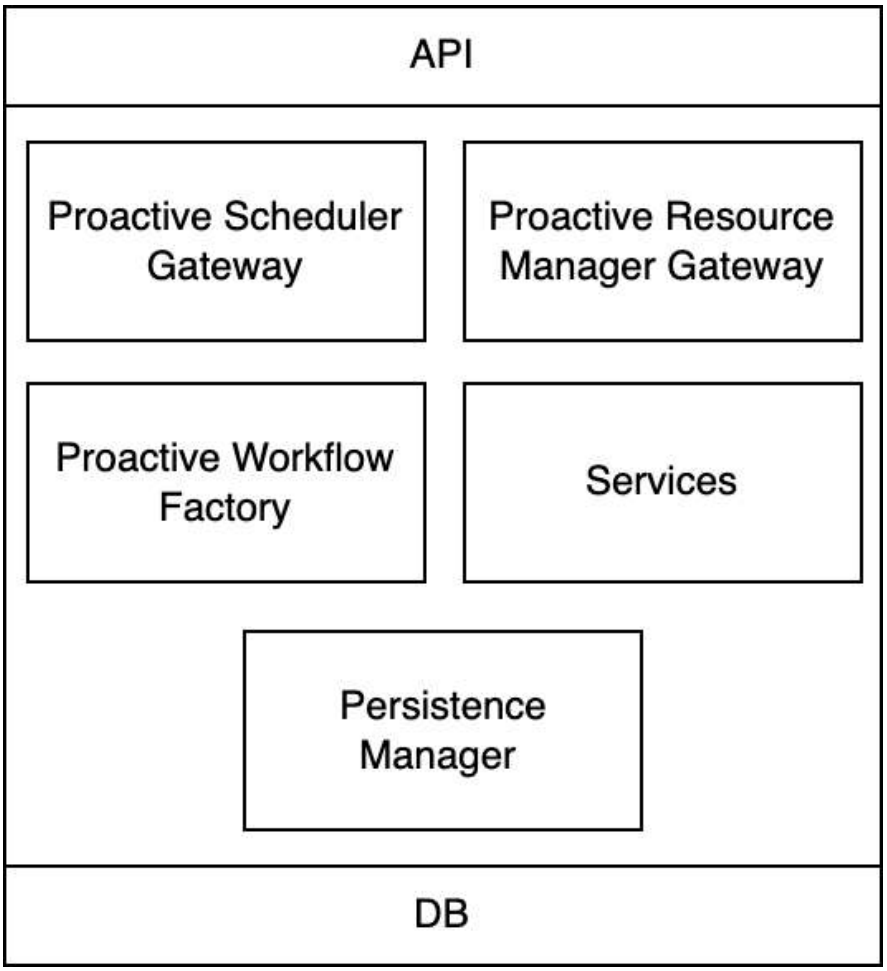

Figure 6 - The SAL's internal architecture

[52] https://gitlab.ow2.org/melodic/morphemic-preprocessor

5.2 Endpoint Presentation

As already indicated, there are two core integration endpoints in SAL that are related to the scope of this deliverable. The first endpoint allows the user to add clouds to the nodes available in the Resource Manager (RM). It has the following signature:

- addClouds(JSONArray clouds): This endpoint takes as input a JSON array that includes information about a set of cloud providers / clouds which need to be added to Resource Manager. These cloud providers are described similarly to the way the *Cloud* class has been modelled in the class diagram of Section 3.1.1, including a subset of the modelled attributes, such as *cloudId*, *cloudProviderName*, *credentials*, *cloudType*, and *endpoint*. The main rationale is that the user supplies the minimal but sufficient information that characterises each cloud such that the Activeeon's RM can then derive the rest of the required information per cloud based on the collection mechanism that will be analysed in the next section.

To retrieve node candidates from the DB by considering respective constraints on resources (given as input and originating from the application's CAMEL model) SAL offers the second integration endpoint. This endpoint has the following signature:

- findNodeCandidates(List<Requirement> requirements): This endpoint takes as input a list of requirements via which all the collected node candidates, residing in the SAL's DB, are filtered. As a result, a filtered list of node candidates is returned from the calls of this endpoint. The requirements include constraints on various characteristics about resources, such as the provider offering them, their image, location, node type and operating system. All of these requirements are collected from the requirement sub-model in the application's model in CAMEL while the characteristics that they encompass directly relate to the attributes and properties of the NodeCandidate class that has been modelled in the class diagram of Section 3.1.1. The filtering of node candidates is conducted via a specific mechanism that is explained in the last section of this chapter.

Both endpoints are synchronous/blocking. This means that the callee blocks until the respective call is ended and a respective result can then be potentially handed over. While this is clear for the second endpoint, i.e., that the callee needs to wait in order to obtain the respective results (which will be logically speaking delivered fast), it is not so clear for the first endpoint as it can be assumed that such a call can take an enormous amount of time, which is not so logical. As it will be explained next, the first endpoint is synchronous in the sense that it registers the need to add one or more cloud providers. The actual addition of these providers and their node candidates happens afterwards in a completely transparent manner.

5.3 Resource Collection Process

Activeeon has developed a particular resource collection and storage process, initiated by the call to the addClouds integration endpoint, which is detailed in D5.3. In a nutshell, this process comprises two steps for the population of the SAL database per cloud provider: (a) first, the respective cloud is added by creating a particular node source in the SAL's RM API as well as an infrastructure controller at the cloud provider side. Further, an instance of the Cloud class is created that will be eventually stored in the SAL's DB; (b) second, a set of requests are made from the node source to the infrastructure controller (via the use of various endpoints) in order to discover and retrieve the rest of the information about a cloud provider as well as all the information about all node candidates offered by the respective cloud provider. That information is then stored in the SAL database.

In the following, we technically detail what happens during the second step of this process. Once addClouds execution is over, SAL will automatically call the endpoint: updateNodeCandidatesAsync (List<String> newCloudsIds) in order to complete the needed information, in the background, and thus register the cloud provider and its offered node candidates in the SAL's database. This endpoint works in an asynchronous mode and executes the following tasks per cloud provider (i.e., for each cloud id available in the endpoint's input parameter):

- Retrieve from the SAL DB the Cloud entity previously created
- From that entity, retrieve the list of the black listed regions
- Using the getImages IaaS-connector endpoint (various IaaS endpoints have been developed by Activeeon to support this feature – all these are supplied by the infrastructure controller), it creates a list of all the available cloud images in the RM excluding the ones that belong to the blacklisted regions
- Using another IaaS endpoint, getNodeCandidates, it creates a list of node candidates that have the same image requirements and cover the same regions

- Finally, a cartesian product is made and node candidates offers related to a provider, a hardware, an image, a location, a node type and an operating system are produced.

5.4 Resource Discovery Process

Once a certain time period after the call to addClouds is made, the information about the required clouds and all their node candidates will be added to the Scheduling Abstraction Layer (SAL) database such that this information will be always available without directly interacting with the cloud providers. Such information can then be exploited in order to support resource discovery through the use of the second integration endpoint, findNodeCandidates. That endpoint incorporates a specific process/algorithm which will be now analysed.
The discovery process is quite simple and straightforward. First, a query is made to the underlying DB in order to obtain all the node candidates from all the cloud providers. Then, for each node candidate, all the requirements given as input to the called endpoint are applied. If this candidate passes this stage, it is added to the final list of filtered node candidates to be returned. More detail on this process/algorithm can be found here[53].

6 Conclusions & Future Work

6.1 Conclusions

This document moved into two directions. On one hand, it explicated the representation model of open-source software that can be used to share related knowledge between the different components of the MORPHEMIC platform and especially those of the Application Profiler module. This representation model has been produced as an extension of the well-known DOAP RDF Schema with a special focus on correlating software in composition hierarchies and specifying the non-functional features of software. The latter non-functional description, although not part of the open-source software repositories from which software metadata are extracted, will be derived by applying static code analysis techniques, which have been thoroughly reviewed in this deliverable and the most promising ones have been selected. This direction has been closed by presenting complementary information about the Knowledge Base, a special component of the Application Profiler module dedicated to the sharing, updating and querying of the extracted open-source software knowledge. This complementary information concerned mainly the API that is exposed by this component that facilitates the interaction with other Application Profiler components as well as the proper management of the open-source software metadata. Complementarity here is denoted with respect to other information that has been already presented in other deliverables, especially the D3.1 one.
On the other hand, the representation model for resources and infrastructures has been presented, which has been devised by Activeeon and applied in its ProActive product suite. This model is coupled with the ProActive API, also presented in this deliverable, which enables performing various resource metadata management activities like metadata querying and access. This coupling enables to have an exact knowledge (with respect to information structure) about which will be the input or output in terms of specific API methods that perform metadata management tasks. This deliverable also shed light on the underlying architecture of this API and especially on the way resource metadata knowledge is extracted and stored. Multiple components from the MORPHEMIC platform are the main exploiters of this API: apart from ProActive itself, the CP Generator can leverage the API in order to search and retrieve the node candidates for each component of a user application while the UI can present various details about available infrastructures and resources to application devops so as to assist them in, e.g., the specification of the right constraints over resources in the CAMEL model of their applications.

6.2 Future Work

The work already conducted in WP1 and T1.3 has paved the way for a more detailed resource and open-source software modelling and analysis. However, we foresee that there are still multiple ways such work needs to be improved or enhanced in order to: (a) derive extra, added-value knowledge that can lead to a better and more accurate deployment reasoning process, a pre-requisite for adaptive, multi-cloud application provisioning; (b) improve the efficiency of the resource metadata collection processes in order to support demanding in terms of filtering performance functionality like the envisioned component grouping one. These ways are shortly analysed in this document in order to set off the respective work needed. Concrete solutions and implementations of these ways will be incarnated in the next and final version of this deliverable (D1.4).

[53] https://gitlab.ow2.org/melodic/morphemic-preprocessor/-/blob/morphemic-rc1.5/scheduling-abstraction-layer/src/main/java/org/activeeon/morphemic/PAGateway.java

6.2.1 Software quality knowledge production

While the selection of the right code analysis tools is the right pathway towards deriving the software quality and security levels, there is still some work missing in terms of selecting the right metrics to use from the selected tools so as to compute these levels. Further, the selection approach in Section 4.2 needs to be also implemented properly and be integrated within the Application Profiler module. Finally, as security is a major issue on its own, various approaches and tools [30] focusing on examining the security levels of software have been already proposed. As such, we will study whether the currently selected tools have a good coverage of the needed security aspects – if this does not hold, then we will review and select additional static analysis tools that have a clear focus solely on the security of software.

6.2.2 Non-Functional resource profiling

The second direction of work focuses on deriving extra knowledge (of profiling type) for the collected resources that can assist in their further filtering. This has a twofold advantage: first, we can capture extra, pragmatic aspects of non-functional resource performance and second, we can exploit such knowledge in order to better satisfy the user requirements as well as produce a more constrained solution space that can benefit and speed up the application deployment reasoning process. The derived profiling knowledge about resource availability and security could play a dual role in MORPHEMIC:

1. resource constraints could be specified in CAMEL models in order to enable the further filtering of resources/node candidates per each application component so as to reduce the enormous solution space
2. resource metrics and partial utility functions could be expressed in these models in order to produce enriched overall utility functions that lead to a further optimisation of the application deployment

To this end, we believe that the two aspects that can be easily covered and can have a great positive impact on user experience are availability and security.

Availability is of paramount importance as the less is the level of resource availability, the lower will be the availability of the application and this can lead to customer frustration and economic loss due to the penalties and other compensation actions that need to be conducted in the context of SLA violations. The main idea that is put on the table concerning this aspect is the derivation of resource availability levels based on the resources usage history. In particular, it is envisaged that the MORPHEMIC platform could rely on an intrusive monitoring approach by exploiting the facilities of the EMS in order to derive a non-functional profile over resources especially on metrics like raw or average availability. For instance, EMS could probe resources while they are being utilised according to a set of specific time periods and then produce an overall raw availability value based on the well-known formula of uptime/totalObservationTime. Then, by aggregating raw availability measurements over a larger time period, average availability values could be produced.

Ideally, apart from resource monitoring, which could be limited to the scope of applications that exploit the MORPHEMIC platform, it will be investigated whether external sources of monitoring knowledge could be leveraged, e.g., in form of potentially existing Web APIs. This could enable to produce the needed, non-functional knowledge about those resources which are, e.g., never utilised by an application. Please note that it could be possible that apart from availability, other non-functional aspects could be derived in this way.

Security has been already considered as one of the most critical factors that impede the cloud adoption for organisations and individuals [31]. To this end, there is a need for adopting a certain approach towards deriving the security level of resources in order to allow their informed selection by users and the eventual increase in the trust that users have in the cloud. There could be different ways to solve such a research problem. One of them concerns the detection of VM or image vulnerabilities [32] similarly to the way statistical analysis on software components enables to derive the components vulnerabilities. Then, based on the detected vulnerabilities, two different methods are envisaged to derive the security profile of resources. One is to just attempt to derive an overall security level of a resource based on the criticality level of the discovered vulnerabilities. Thus, a kind of a security score can be derived and be associated with cloud resources.

The second method is to seek into correlated vulnerability knowledge (based on standards like CVE[54] and CWE[55]) and attempt to check the absence of security controls. In other words, we seek whether each vulnerability can lead to an attack that can damage certain security aspects that indicate either the lack of a security control or an improper or inadequate implementation of that control (e.g., due to a misconfiguration). In this respect, the resource profile is

[54] https://www.cve.org/
[55] https://cwe.mitre.org/

populated with negative knowledge about security control support and such a knowledge can be utilised for resource filtering purposes.
We foresee, and this is our ultimate goal, that through a more complete and continuous knowledge about cloud resources, the prediction techniques that are proposed in MORPHEMIC could be exploited in order to predict the performance of cloud resources and enable a more informed decision-making at the infrastructure level by, e.g., moving application components from one cloud resource to another, if the first will fail or the second leads to a better performance with a similar or lower cost. Thus, this is something that is worth being investigated once proper, non-functional knowledge about resources is put in place.

6.2.3 Resource metadata collection and updating

Currently, while the Executionware attempts to collect and store the resource metadata knowledge about the different clouds that are supported, the Upperware is on hold. This creates a long waiting time and can significantly delay the initial deployment of a multi-cloud application. To this end, there is a need to invent one or more strategies that can be followed in order to either speed up this process or smartly organise it such that the initial delay that can substantially decreased. One strategy that could be followed is to parallelise the processing of each cloud through the use of a multi-threading approach. Another strategy would be to allow the initial application deployment when a specific percentage or amount of node candidates has been already produced. Yet another strategy could be to cover some representative cases of node candidates first. Another promising strategy could be to produce some constraints from the application's CAMEL model and use them as an on-the-fly filtering criterion over the resource metadata during their collection.
The use of a strategy comes with its own advantages or disadvantages. For example, by considering the last strategy from the above, it could be argued that we produce quickly a sufficient node candidate set that will be always utilised for the application deployment, thus not including resources that will be never exploited. On the other hand, such a node candidate set reduction is application-specific and not flexible enough to accommodate for the change – change in user requirements or change in the cloud provider's offering portfolio. As such, the trade-offs between the use of the different strategies need to be accounted for before the final selection is conducted. The existence of such a trade-off argues about the need to combine strategies like the ones already mentioned in such a way that could enable to retain the advantages of the combined strategies as well as alleviate or diminish their disadvantages. For instance, we could choose to parallelise the collection from the different cloud providers as well as apply the on-the-fly filtering in a setting where change is not so frequent.
While speeding up the collection of the resource metadata is one research problem, there is an additional one concerning the updating of such metadata in the course of time. This updating is essential for adaptive systems which are able to sense changes in their environment and react on them. In the context of resource metadata, such an updating can reveal new opportunities for optimisation, e.g., in case new resource offers are advertised by the cloud providers. It can also invalidate previous deployment reasoning decisions and solutions when they, e.g., rely on resource offers which do not hold any more or quite soon will become outdated. As such, there is a need for a smart resource metadata updating strategy that does not overwhelm the platform and is able to sense the changes as soon as they occur. Overwhelming the platform can be easily achieved via a very frequent updating process, which aims to sense the changes as soon as possible, in conjunction with the fact that such a process is heavy as it requires communicating with multiple cloud providers and processing the respective information drawn. To this end, it needs to be decided which is the right frequency for such an updating and whether there can be a smart mechanism that can be somehow informed of the changes when they actually occur such that there is no need for any kind of periodic updating. In case the latter is not possible, then one solution (updating strategy) that could be designated is to support a distributed updating process with the right updating frequency which attempts to operate in each separate cloud from the ones supported and then cause the updating of the resource metadata in the centralised database only when it is actually needed. As the number of records to be updated would be significantly small and could affect only a single cloud with a very high probability (not all clouds update their offerings at the same time), the DB updating would be rather fast. In fact, the distribution is not only an exclusive strategy of the updating process but could be an efficient solution also for the collection process.
In any case, this is a direction of work of top priority and the involved partners in WP1 will contribute the appropriate research and development effort towards both deriving, combining and evaluating alternative (resource metadata collection & updating) strategies such that the best / optimal one is selected for the realisation in the MORPHEMIC platform within the auspices of the Executionware.

6.2.4 Resource storage structure and filtering

It has been well argued in the literature that the way information is stored can have a great impact on its querying and filtering. This is also inevitably true for the case of resource metadata storage, querying and filtering where different storage structures and filtering algorithms have been proposed, each coming with its own pros and cons. Here we need

to stress the fact that resource filtering is of paramount importance for the deployment reasoning process for various reasons. First, it enables to reduce the solution space by filtering the node candidates that can support the deployment of each application component. Second, it will play an important role in the envisaged functionality of component grouping envisaged. Component grouping will examine different but legitimate (in terms of user constraints) and compatible ways via which application components can be grouped together in order to be considered as a whole (i.e., a new composite component) during application deployment. Such ways need also to be realistic in the sense that they can be accommodated by the overall solution space. For instance, placing three components in one VM creates the need that the VM characteristics are sufficient in order to support such a placement. Also, if the components are executed concurrently, then there is a need to check whether their aggregated resource requirements (e.g., total number of cores, total amount of main memory) can be met by the (VM) resource capabilities. As such, while producing all possible compatible combinations of application components, there is a need to quickly filter the collected node candidate set in order to check that it does map to at least one cloud (service) offering for each examined combination. Based on this rationale, it is evident that resource metadata filtering should be conducted as quickly as possible in order not to prolong significantly the already planned component grouping functionality.

However, the current storage structure and filtering mechanism utilised are not so efficient in this aspect. While using a relational database enables to support transactionality and to support fast query processing, such a database kind is not suitable in terms of performance when complex queries need to be posed with multiple joins including multiple tables. Such complex queries, however, are the norm for supporting proper resource filtering due to the existence of multiple classes that should be taken into account and that have elements which participate in the resource constraints that need to be met (see also class diagram in Section 3.1.1). As each class maps to a specific table, then obviously the respective query would need to become complex. While the construction of a complex query is not involved in the logic of the current filtering process, the approach followed for the filtering is quite simplistic and not efficient at all. It relies on using a simple query which attempts to fetch all node candidates while the number of such candidates would be quite high (in the order of tens of thousands or even higher orders). Even worse, it then attempts to process each node candidate individually and apply the respective constraints in order to filter it. The respective time complexity is thus O(N), something that needs to be avoided for very high values of N (where N means the number of candidates).

Fortunately, the available literature includes various approaches that can be followed in order to address the above problem. For instance, in the web service world, specialised structures (e.g., ordered sets per filtering dimension [33] have been devised that enable to support an ultra-fast matchmaking/filtering process. By combining these structures with appropriate storage means like distributed object databases and/or memory object caching systems (e.g., memcached), a more efficient system could be produced able to satisfy the needs for a fast resource filtering and component aggregation processes. One could imagine even a hybrid approach storage approach where the specialised structure is kept on distributed memory and is constantly synchronised with the underlying (relational) DB, which still needs to be maintained as it is utilised also for resource provisioning purposes. Such a solution is inspired by work in MELODIC (thus inherited by MORPHEMIC) where a distributed memory (Memcached) is utilised for storing the filtered node candidates for their fast retrieval by the relevant components involved in the application deployment reasoning process.

It should be noted here that the use of a distributed storage approach could also enable to support distributed filtering in the sense that we could keep cloud-provider specific node candidates in the node that hosts also the provider-specific infrastructure controller. As a similar amount of node candidates maps to each public cloud provider, both the storage and filtering load could be distributed evenly.

7 References

[1] Amir Taherkordi, Ciro Formisano, Geir Horn, Kyriakos Kritikos, Maria Antonietta Di Girolamo, and Marta Różańska, "D3.1 Software, tools, and repositories for code mining," MORPHEMIC Project Deliverable, Dec. 2020.

[2] D. Baur, D. Seybold, F. Griesinger, H. Masata, and J. Domaschka, "A Provider-Agnostic Approach to Multi-cloud Orchestration Using a Constraint Language," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Washington, DC, USA, May 2018, pp. 173–182. doi: 10.1109/CCGRID.2018.00032.

[3] Marta Różańska *et al.*, "D4.1 Architecture of pre-processor and proactive reconfiguration," MORPHEMIC Project Deliverable, Dec. 2020.

[4] A. P. Achilleos *et al.*, "The cloud application modelling and execution language," *J Cloud Comp*, vol. 8, no. 1, p. 20, Dec. 2019, doi: 10.1186/s13677-019-0138-7.

[5] D. Palma and T. Spatzier, "Topology and Orchestration Specification for Cloud Applications (TOSCA)," Organization for the Advancement of Structured Information Standards (OASIS), Jun. 2013. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cos01/TOSCA-v1.0-cos01.pdf

[6] F. Moscato, R. Aversa, B. D. Martino, T.-F. Fortis, and V. I. Munteanu, "An Analysis of mOSAIC ontology for Cloud Resources annotation," in *Federated Conference on Computer Science and Information Systems - FedCSIS 2011, Szczecin, Poland, 18-21 September 2011, Proceedings*, 2011, pp. 973–980. [Online]. Available: http://ieeexplore.ieee.org/document/6078209/

[7] G. G. Castañé, H. Xiong, D. Dong, and J. P. Morrison, "An ontology for heterogeneous resources management interoperability and HPC in the cloud," *Future Generation Computer Systems*, vol. 88, pp. 373–384, Nov. 2018, doi: 10.1016/j.future.2018.05.086.

[8] A. Zhou, K. Ren, X. Li, W. Zhang, and X. Ren, "Building Quick Resource Index List Using WordNet and High-Performance Computing Resource Ontology towards Efficient Resource Discovery," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Zhangjiajie, China, Aug. 2019, pp. 885–892. doi: 10.1109/HPCC/SmartCity/DSS.2019.00129.

[9] Kais Chaabouni *et al.*, "D1.1 Data, Cloud Application & Resource Modelling," MORPHEMIC Project Deliverable, Dec. 2020.

[10] D. K. Nguyen, F. Lelli, Y. Taher, M. Parkin, M. P. Papazoglou, and W.-J. van den Heuvel, "Blueprint Template Support for Engineering Cloud-Based Services," in *Towards a Service-Based Internet*, vol. 6994, W. Abramowicz, I. M. Llorente, M. Surridge, A. Zisman, and J. Vayssière, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 26–37. doi: 10.1007/978-3-642-24755-2_3.

[11] C. Quinton, D. Romero, and L. Duchien, "Cardinality-based feature models with constraints: a pragmatic approach," in *SPLC 2013: 17th International Software Product Line Conference*, 2013, pp. 162–166.

[12] Daniel Baur and Daniel Seybold, "D4.1 Provider agnostic interface definition & mapping cycle," Melodic Project Deliverable, Sep. 2019.

[13] M. Atzeni and M. Atzori, "CodeOntology: RDF-ization of Source Code," in *The Semantic Web – ISWC 2017*, vol. 10588, C. d'Amato, M. Fernandez, V. Tamma, F. Lecue, P. Cudré-Mauroux, J. Sequeda, C. Lange, and J. Heflin, Eds. Cham: Springer International Publishing, 2017, pp. 20–28. doi: 10.1007/978-3-319-68204-4_2.

[14] J. Lehmann *et al.*, "DBpedia – A large-scale, multilingual knowledge base extracted from Wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015, doi: 10.3233/SW-140134.

[15] A. Alnusair and T. Zhao, "Component search and reuse: An ontology-based approach," in *2010 IEEE International Conference on Information Reuse & Integration*, Las Vegas, NV, USA, Aug. 2010, pp. 258–261. doi: 10.1109/IRI.2010.5558931.

[16] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Science of Computer Programming*, vol. 79, pp. 241–259, Jan. 2014, doi: 10.1016/j.scico.2012.04.008.

[17] Yih-Fam Chen, E. R. Gansner, and E. Koutsofios, "A C++ data model supporting reachability analysis and dead code detection," *IIEEE Trans. Software Eng.*, vol. 24, no. 9, pp. 682–694, Sep. 1998, doi: 10.1109/32.713323.

[18] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, and others, *Web services description language (WSDL) 1.1*. 2001.

[19] M. J. Hadley, "Web application description language (WADL)," 2006.

[20] O. Coalition, *OWL-S 1.0 Release*. DAML, At http://www. daml. org/services/owl-s/1.1, 2005.

[21] D. Roman *et al.*, "Web service modeling ontology," *Applied ontology*, vol. 1, no. 1, pp. 77–106, 2005.

[22] D. Palma, M. Rutkowski, and T. Spatzier, *TOSCA Simple Profile in YAML Version 1.0*. 2015.
[23] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut, "Quality of service for workflows and web service processes," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 3, pp. 281–308, 2004.
[24] J. Amsden, "Modeling with soaml, the service-oriented architecture modeling language–part 1–service identification," *IBM Developer Works, http://www. ibm. com/developerworks/rational/library/09/modelingwithsoaml-1/index. html*, 2010.
[25] M. Dimitrov, A. Simov, S. Stein, and M. Konstantinov, "A BPMO Based Semantic Business Process Modelling Environment," 2007.
[26] K. Kritikos *et al.*, "A survey on service quality description," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, p. 1, 2013.
[27] J. R. Chromy, "Encyclopedia of Survey Research Methods," Thousand Oaks: SAGE Publications, Inc., 2019. doi: 10.4135/9781412963947.
[28] T. L. Saati, *The Analytic Hierarchy Process*. McGraw-Hill, 1980.
[29] Łukasz Szymański, Ali Fahs, Maroun Koussaifi, Chris Kachris, Mohamed Khalil Labidi, and Paweł Skrzypek, "D5.3 Deployment Artefact Manager," MORPHEMIC Project Deliverable, Jun. 2021.
[30] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *2014 IEEE Symposium on Security and Privacy*, San Jose, CA, May 2014, pp. 590–604. doi: 10.1109/SP.2014.44.
[31] W. A. Jansen, "Cloud Hooks: Security and Privacy Issues in Cloud Computing," in *2011 44th Hawaii International Conference on System Sciences*, Kauai, HI, Jan. 2011, pp. 1–10. doi: 10.1109/HICSS.2011.103.
[32] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro, "A security analysis of amazon's elastic compute cloud service," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, Trento, Italy, 2012, p. 1427. doi: 10.1145/2245276.2232005.
[33] K. Kritikos and D. Plexousakis, "Novel Optimal and Scalable Nonfunctional Service Matchmaking Techniques," *IEEE Trans. Services Computing*, vol. 7, no. 4, pp. 614–627, 2014, doi: 10.1109/TSC.2013.11.