D3.1 Software, tools, and repositories for code mining

MORPHEMIC

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
D3.1

Date
31 December 2020

Responsible partner
Engineering Ingegneria Informatica S.p.A.

Editor(s)
Maria Antonietta Di Girolamo

Reviewers
Yiannis Verginadis, Sebastian Geller

Distribution
Public

Availability
www.MORPHEMIC.cloud

Executive summary

This document provides a detailed description of the software and the tools to be used for code mining. In the MORPHEMIC project, Code mining is needed to define application profiles, to be used for better adapting the available polymorphic deployment configuration to the requirements specific of the application. The applications' deployment models provided by MORPHEMIC must be dynamic and adaptive, and capable of handling any expected or unexpected situation. In this way MORPHEMIC assists, the application to supply a more or less constant level of service. The Polymorphic Adaptation works at both the architecture and cloud service level, by defining the most optimal deployment model according to internal (e.g., available infrastructures) and external (e.g., load) constraints. This means that the code mining functionality helps to define an application profile, which, in turn, is used to obtain the best possible adaptation of the application deployment to the available infrastructures and component configurations/forms. The process of Code Mining in MORPHEMIC is composed by three tasks: web crawling, code analysis and data storage. The three aforementioned tasks define the three components on which this deliverable is focused. In particular, the web crawler has been identified among some candidates in order better support the extraction of sets of information associated with projects available on known source repositories (for example GitHub). A prototype of the Web Crawler has been implemented and running. It is based on the outcome of an EU co-funded research project: MARKOS. The MARKOS' Web Crawler was analyzed, and its architecture was modified to be more suitable for MORPHEMIC. The second part of the deliverable provides an analysis of the open source code repositories to be used by the MORPHEMIC's Web Crawler to get data and metadata. The Knowledge Base, implementing the functionality of data storage, has been designed and includes part of the functionalities provided by the MARKOS' Web Crawler. Finally, the Code Analyser should be still selected among some candidates. The last part of the deliverable provides the results of an analysis on code classification. The various types of projects identified by code mining can be considered as sources for code classes such as High-Performance Computing or web code. The first step of the analysis concerns the identification of a set of techniques and tools for code analysis. The second step goes deeper to explore the concept of code classification, including appropriate methodologies to enable the recovery of optimal deployment patterns.

Author(s)

Amir Taherkordi (UiO), Ciro Formisano (ENG), Geir Horn (UiO), Kyriakos Krytikos (FORTH), Maria Antonietta Di Girolamo (ENG), Marta Różańska (UiO).

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

Revisions

Date	Version	Partner	Description
04/11/2020	Draft (0.1)	ENG, FORTH, UiO	1st draft version
11/11/2020	Draft (0.2)	ENG, UiO	Update Section 2, Section 4, Section 6
16/11/2020	Draft (0.3)	ENG, UiO	Updating figure 1, Figure 2, Figure 3, add Figure 4, updating Section (from 2 to 6) after merging section 2 with section 4 (renamed section 2.2.2.2), Section 5
18/11/2020	Draft (0.4)	ENG, UiO, FORTH	Updating section 2 and section 5.
30/11/2020	0.5	ENG	Complete version ready for the first review
07/12/2020	0.6	ICCS	1st Review comments
22/12/2020	0.7	ENG	Complete version ready for the second review
05/01/2021	0.8	ICON	2nd Review comments
22/01/2021	0.9	ENG, UiO, FORTH	Revised version
24/01/2021	1.0	ENG	Ready for release version
10/02/2021	1.1	ENG, FORTH, UiO	Updating section 5.2.4, section 2.1.3 and section 6
15/02/2021	1.2	ENG	Revision of the document

Table of Contents

Index of Figures

Index of Tables

Glossary

ABBREVIATIONS	
AI	Artificial Intelligence
API	Application Programming Interface
CFGs	Control Flow Graphs
CP	Constraint Problem
CRAN	Comprehensive R Archive Network
CTAN	Comprehensive TeX Archive Network
DNA	Deoxyribonucleic acid
DOAP	Description of A Project
ETL	Extract Transform Load
EMS	Event Management System
HPC	High Performance Computing
JSON	JavaScript Object Notation
LCS	Longest Common Subsequence
LDA	Latent Dirichlet Allocation
MARKOS	MARKet for Open Source - An Intelligent Virtual open source Marketplace
ML	Machine Learning
MPL	Mozilla Public License
N/A	Not Available
OSP	open-source Project
PMC	Project Management Committee
PDGs	Program Dependence Graphs
RDF	Resource Description Framework
RNA	Ribonucleic Acid
REST	Representational State Transfer Application Programming Interface
SOTA	State-of-the-Art
SQL	Structured Query Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
XML	eXtensible Markup Language
WWW	World Wide Web

1. Introduction

1.1 Scope

This deliverable describes the software and the tools for code mining that will be part of the architecture of the MORPHEMIC platform[1].
Code mining is conceptually a *data mining* task focused on code, i.e., the process of extracting appropriate *code* from code repositories. This means that the code and the associated metadata should be *found*, *stored* and *analysed*; therefore, the concept of code mining in MORPHEMIC consists of three activities:

- *crawling,* to search for useful code from external repositories;
- *storage,* to store code and metadata;
- *code analysis,* to extract useful information (e.g., features) from the code found.

The status of design and implementation of the tools implementing these functionalities is different. Specifically, the functionalities of crawling and storage will be available in the first release of MORPHEMIC. The basis on which the tools have been built has been defined after a technological analysis whose results will be presented in this document. Specifically, the basis of both the tools is the Web Crawler produced in the MARKOS project[2]. The first release of MORPHEMIC will include a Web Crawler providing similar structure and some minor modifications with respect to MARKOS' Web Crawler. More improvements are planned for the next releases that will also provide an autonomous version of the Knowledge Base, currently included in the Web Crawler. The main conceptual difference, that will drive all the next phases of the development process of the MORPHEMIC's Web Crawler, is that the output of the crawling process will be evaluated against the objectives of the Polymorphic and Proactive adaptation, while, in the case of MARKOS the target was the evaluation of licences.
Polymorphic Adaptation is one of the pillars of MORPHEMIC and includes the functionality of application profiling. Specifically, Polymorphic Adaptation allows applications to be deployed on different environments (including multi-cloud, edge, fog) and change their configurations based on the application features and context in order to maximize the relevant advantages (e.g., application performance).
In this context, the definition and architecture of the Application Profiler is supplied (Section 2).
Concerning the code analysis, at the time of writing this deliverable, a first evaluation of the tools implementing this functionality has been performed along with an analysis of the techniques, methodologies and algorithms for code classification.

1.2 Intended Audience

The intended audience of this deliverable is:

- *MORPHEMIC Use Case partners* who need to have a clear view of the tools, repositories, algorithms and techniques used for code mining as well as an understanding on how code mining results can benefit the development and optimisation of their use-cases applications
- *MORPHEMIC developers, technicians, administrators and researchers* involved in the implementation and integration of the various components implemented for the MORPHEMIC platform, in particular for the Profiler components.
- *MORPHEMIC researchers* involved in, for example, the polymorphic adaptation activity or in the CAMEL modelling framework.
- *MORPHEMIC adopters and external researchers* that would like to contribute to the open source *MORPHEMIC code* after the end of the project; to other external users with specific interests, such as code quality (e.g., BetterCodeHub[3]).

[1] https://www.morphemic.cloud/
[2] https://cordis.europa.eu/project/id/317743
[3] https://www.bettercodehub.com/

1.3 Document Organization

The current chapter introduces the scope, the objectives, and the structure of this document. The last chapter is dedicated to the final considerations and the planned next steps. The remaining chapters are the following:

- *Chapter 2 ("The Application Profiling")* introduces the concept of application profiling as well as the high-level architecture of the Application Profiler and its components.
- *Chapter 3 ("Code mining components")* is focused on the three aforementioned components providing the functionality of code mining and their contribution to application profiling.
- *Chapter 4 ("Evaluation of the Crawling process")* provides an evaluation and a justification of the technological choices for the crawling process: specifically, the Web Crawler component will be evaluated against the objectives of MORPHEMIC, while the source code repositories associated or associable will be analysed in terms of the provided information.
- *Chapter 5 ("Algorithms for classifying the code")* reports the state of art about the tools and algorithms for the code analysis and classification which could be selected, potentially extended and finally applied within the MORPHEMIC project.

2. The Application Profiling

2.1 Introduction

This chapter includes an analysis of the application profiling functionality, that is the generic context in which the code mining functionality provides its contribution. From a technical point of view, the three modules of code mining, mentioned in the previous section, are part of the Application Profiler, i.e., the MORPHEMIC module responsible for maintain the profile of a polymorphic application. For this reason, it is very important to define the functional and technological context in which the code mining tools are seen: this is the objective of this chapter.
The Polymorphic (Application) Adaptation feature focuses on supporting the deployment and reconfiguration of polymorphic applications, i.e., applications that change their architecture (variant) at runtime by selecting a different application component form from those possible based on their requirements and context.
The design and development of an "*Application Profiler*" is one of its main tasks: specifically, an Application Profiler is useful to determine the best deployment options available for each variant of the application architecture. In order to define and maintain a suitable Application Profile, it is very useful to find the software profiles of similar applications (i.e., applications that deliver similar or equivalent functionality with respect to that of an application component): on this sense, the analysis of data and metadata of publicly available projects helps.
Therefore, code mining is one of the core functionalities of the Application Profiler module, part of the Polymorphic Adaptation feature. Figure 1 Polymorphic Adaptation architecture. shows the overall architecture of the Polymorphic Adaptation feature and highlights the Application Profiler module, which directly interacts with the following elements:

- The external *Knowledge Sources*, used for two main reasons: (a) to suggest and to construct an initial performance model of the application; (b) to crawl and to mine source code repositories. Specifically, external Knowledge Sources can be open source code repositories like GitHub (Section 4.3.1), as well as generic sites.
- The *Event Management System* (*EMS*), for updating the non-functional part of the application profile in order to increase the quality of its precision.
- Communication with the *User Interface* (*UI*) follows different paths and enables the Application Profiler to obtain as input different data (such as the CAMEL model[4]) to construct the application profile and be updated in the event of their modifications.
- The *Constraint Problem* (*CP*) Generator generates the CP Model from the application profile. A CP model is a constraint optimisation model which enables us to reason the best deployment solutions for the current application.
- The *Architecture Optimiser* uses the application profile to assess the need for changing the application architecture variant. Please note that each application architecture variant is derived from a different combination of the forms/configuration classes (e.g., serverless/functions, container, hardware accelerator-based, etc.) of all the application components. Further, a selected application architecture variant is the main input to the MORPHEMIC core platform in order to conduct deployment reasoning and discover the best configuration for the application at hand.

[4] http://camel-dsl.org/

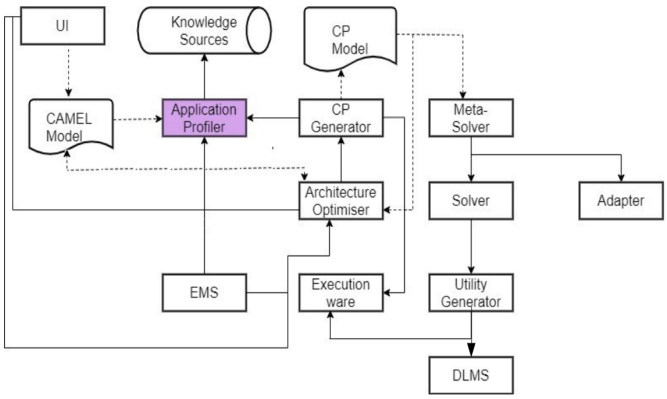


Figure 1 Polymorphic Adaptation architecture.

Therefore, the Polymorphic Adaptation feature relies on a composite component called the *Application Profiler*. This component produces and maintains application profiles that can be used to derive suitable deployment models aiming to optimize the application deployment at both the architecture and cloud service level. The next section provides the definition, role and responsibilities of the Application Profiler.

2.1.1 Application Profiler

The MORPHEMIC project aims to support the polymorphic modelling and adaptation of multi-cloud applications, both in a reactive and proactive manner. Such applications usually comprise a single architecture, in which each application component has a specific form. In this respect, the goal of a multi-cloud management platform is just to find the best possible cloud services at the infrastructure level and support the execution of these fixed-form components. However, to really support the so-called application polymorphism [1], there is a need to create the space for multiple architecture variants which can be created through considering multiple forms for the same application component (which deliver the same functionality). As such, then the goal of the platform is first to find the right form of each application component based on the application requirements and then the best possible infrastructure service for this form.
In this context, the profile of an application should cover all architecture variants of an application as well as the different quality of service levels that can be reached by these variants. The latter information is quite crucial as through the consideration of the user requirements, it can then be possible to select the best possible variant that satisfies them. The management of such a profile is the main responsibility of the Application Profiler module in MORPHEMIC.
The Application Profiler should focus only on the complex task of managing the profile of an application while the CP generator will be responsible to generate different CP models to assist in the application architecture and configuration optimisation tasks by relying on the maintained application profile. In the following, we will focus on the Application Profiler module. A detailed analysis of the Architecture Optimiser of and how the CP Generator contributes to the architecture and configuration optimisation tasks will be supplied in deliverable *"D3.3 - Optimized planning and adaptation approach"* (M16).

2.1.2 Role and Responsibilities of the Application Profiler

The role and responsibilities of the Application Profiler component are related mainly to the construction and maintenance of application profiles. In particular, the Application Profiler assists in the production of different variants of application architecture by finding software components that might have a different form and deliver the same functionality as the application components. The analysis of the open source software, found in code repositories, allows us to define new forms of application components and new deployment models that enhance existing application profiles.
Therefore, the role of the Application Profiler is mainly to construct, enhance and maintain the profile of a polymorphic, multi-cloud application. In the context of this role, we can deduce that the main responsibilities of the Application Profiler component are the following:

- *Application profile construction*: based on the specification of a polymorphic, multi-cloud application in CAMEL, the Application Profiler is able to construct application profiles, covering the functional aspect in

terms of both the component and application level as well as the non-functional aspect in terms of the application (non-functional) requirements (e.g., average application execution time less than 2 hours) and capabilities (e.g., average response time for analytics component less than 1 hour). The latter should cover the construction of the non-functional model per each application component form and application architecture variant. The process of profile construction is depicted in Figure 2 Process of application profile construction.

- *Application profile enhancement*: here the main responsibility of the Application Profiler is to enhance the basic application profile towards mainly the functional aspect. In particular, the main goal here is to support the discovery and incorporation of new forms of application components. This goes down to browsing and searching open source software repositories as well as analysing the knowledge drawn in order to classify the functionality of open source software components and match it against the one exhibited by an application component. As the matched software components can have different forms, this can lead to suggesting new forms of application components that need to be verified so as to be included in the application profile. The process of profile enhancement is depicted in Figure 3.
- *Application profile maintenance*: the profile of the polymorphic, multi-cloud application that has been constructed and enhanced also needs to be maintained. The maintenance can again be split into *functional maintenance* and *non-functional maintenance* based on the two main aspects of focus. Functional maintenance mainly covers the incorporation of new application component forms upon their verification by the DevOps in the application profile as well as the continuous monitoring of the CAMEL application model in order to detect changes in application components and their forms and thus update the application profile. Non-functional maintenance mainly concerns the construction of new performance models for updated or new application components as well as the modification of existing performance models based on the monitoring feedback collected through the execution of the user application.

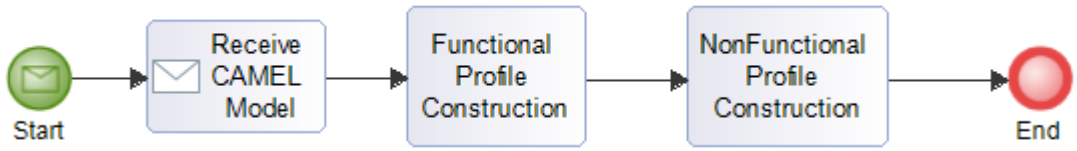


Figure 2 Process of application profile construction

Figure 3 Process of (functional) profile enhancement

2.1.3 Architecture of the Application Profiler

Based on the above analysis concerning the main role and responsibilities of the Application Profiler, a first version of the architecture of this component has been constructed. This architecture can be seen in Figure 4, where the highlighted components are implementing the functionalities related to code mining.

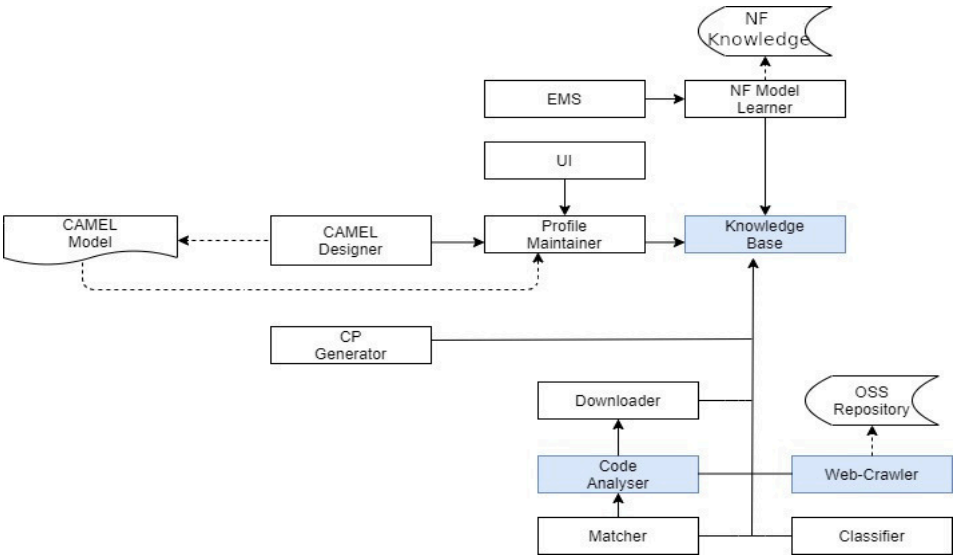


Figure 4: Architecture of the Application Profiler

Specifically, the functionalities and tools related to code mining, i.e., the Web Crawler, the Code Analyser and the Knowledge Base will be described in chapter 3. These functionalities were not present in MELODIC.
The following subsections will provide a brief description of the other components of the Application Profiler. These descriptions will be very useful to better define the context of the Code Mining components.

Downloader

This component is responsible for downloading the source-code of open source software components upon the respective incoming request. The downloaded code can be stored in the local file system, in case this component is situated in the same host as that of the Code Analyser, which requires analysing this code; otherwise, it needs to be stored in the Knowledge Base.

Classifier

The Classifier assists in the enhancement of application profiles by classifying open-source components, which can potentially match application components, according to the functional aspect. The input is the information already crawled and analysed in the Knowledge Base (i.e., metadata and functional features). This means that the input should have already been processed by the Crawler and Analyser components. The collaboration diagram in Figure 5 shows how the Classifier interacts with the Knowledge Base to classify the software and enrich the stored information. The enriched information is then used to support the discovery of new forms for application components. Specifically, the classification concerns a set of functional categories associable to the software and the actual deployment form that they take (with respect to their configuration/form). Such an output also concerns the application components, which could be in turn also classified. This is certainly an activity that facilitates the matching task of the Matcher, as will be described later.
Depending on whether the Classifier applies any kind of machine learning technique, the classification model is another output that can be produced which is also stored in the Knowledge Base. Such a classification model is quite important as it can support both the classification of new components as well as the matching of all processed components with, e.g., application components.

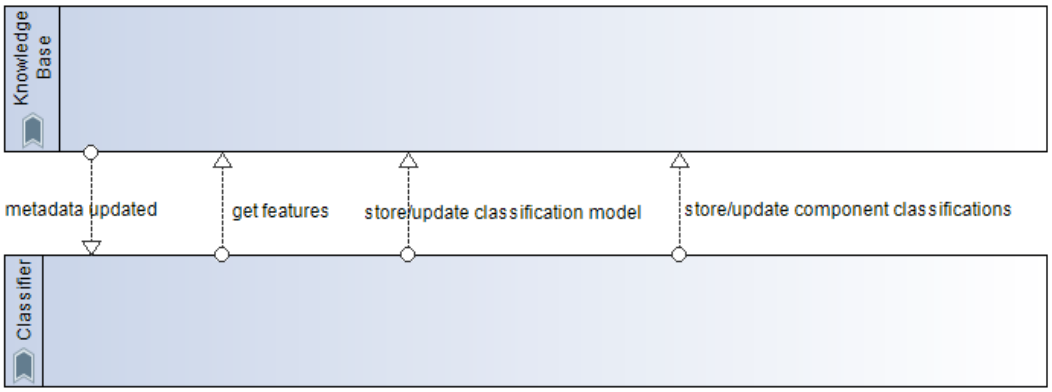


Figure 5 Classification functionality fulfilment through a collaboration diagram

Please, note that different implementations of a certain component might be revealed by applying different classification techniques based on potentially different functional feature forms. The selection of such a technique depends on various factors, such as the kind and number of the software components involved, the domain on which they are specialised and the knowledge available for these components. Potentially, it can then be a user/DevOps decision which classification technique to utilise per each factor (value) combination.
Finally, it is important to highlight that this component not only produces but also maintains classification knowledge. Thus, the Classifier needs to detect when the classification model needs to be updated, modify it as well as update the functional categories that have been associated to both the open source software components already crawled and processed as well as the application components.

NF Model Learner

The *Non-Functional* (NF) Model Learner implements and maintains the non-functional aspects of the application profile. In particular, this component is in charge of constructing and maintaining a performance model for each form of each component of an application as well as each architecture variant of the application itself. The model construction could rely on the execution history of the application components, especially if such components are re-used in the context of other applications. In case such an execution history does not exist, then the NF Model Learner could inspect other sources to derive the needed knowledge, such as benchmarking ones. Both cases are depicted in Figure 6.

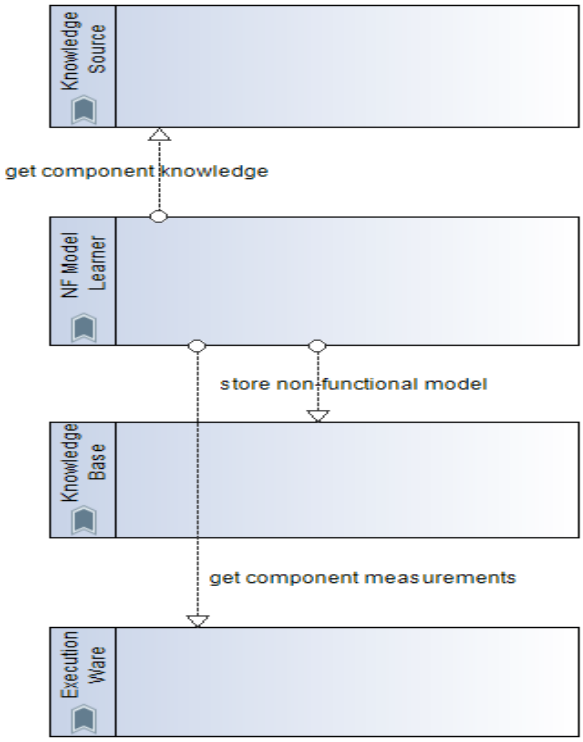


Figure 6 Non-functional profile construction

The performance model is maintained and updated to be as precise as possible based on the execution history of the current application (i.e., based on the measurements collected by the platform while the application is being deployed and executed). Such a model is stored and updated within the Knowledge Base: it is shown in Figure 7.

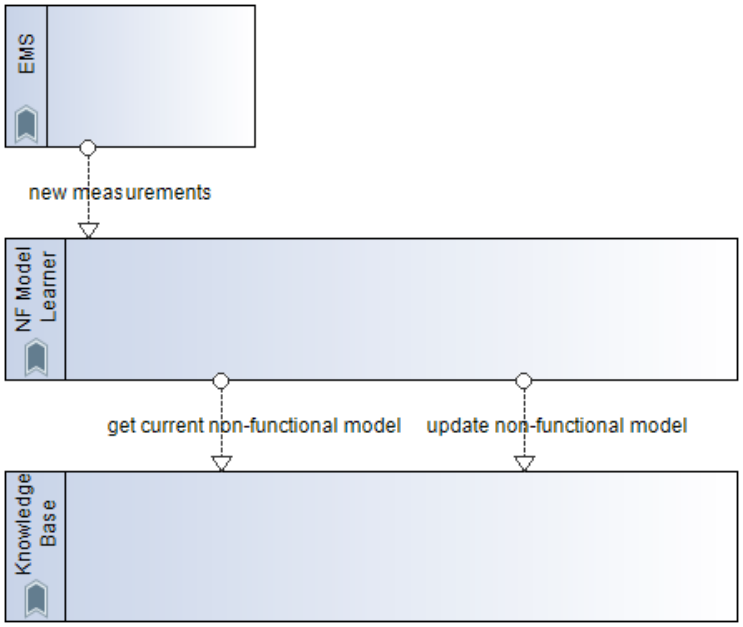


Figure 7 Non-functional profile maintenance

Matcher

The Matcher component is responsible for matching application components and their requirements with open source software components as can be seen in Figure 8 Matching of application components. As such, it exploits the knowledge derived from the CAMEL model in terms of the application components as well as the knowledge stored in the Knowledge Base, derived from the open source software components by the Classifier and the Analyser. The matching will rely mainly on the functional aspect. This means that the components are matched based on their functional categories. This pre-supposes that also 'the application components have been mapped into specific functional categories by the Classifier, as mentioned above.

The produced matching results will be ranked based on the non-functional knowledge (component quality and security levels) derived by the Analyser. The objective of the matching process is to derive new forms of existing application components. This signifies that there is no point in returning results that map to existing application component forms unless the user/DevOps desires to replace them. The output of this component, i.e., the matching results, is associated with the matched application component in the application profile / Knowledge Base. However, it is the duty of the DevOps to test the matching results before approving them to become the new forms of application components. Upon approval, the new forms of application components are also inserted in the CAMEL model of the application and, thus, taken into account in the forthcoming Application Variant Selection / Architecture Optimisation Reasoning process executions (operated in the context of a deployment of the corresponding application).

This component is complementary to the functionalities supplied by the Classifier and Code Analyser. It can actually be stated that the Classifier and Analyser prepare the main ground for supporting the matching task which results in the enhancement of the functional profile of an application.

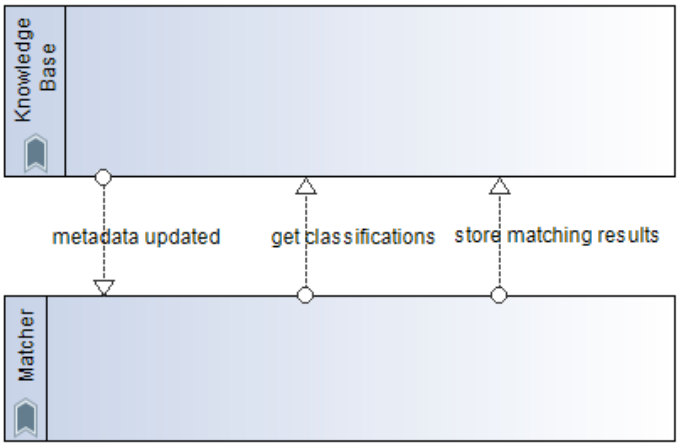


Figure 8 Matching of application components

It must be highlighted that the Matcher associates matching results with the components of all applications handled by the respective MORPHEMIC platform instance. However, it can also be utilised for consulting purposes irrespective of any application already handled by the platform instance. In particular, the Matcher can be considered as a micro-service which enables us to find the right open source software components that match arbitrary components, whose source code URL is supplied as input to the respective method of this micro-service. In that case, the Matcher will need to first discover the functional categories of that arbitrary component before matching them with those of the open source software components crawled. This will require utilising the Code Analyser to extract the functional features of the component to be matched and then applying the classification model, derived by the Classifier and stored in the Knowledge Base, on the extracted features in order to produce the right categories of that component. This matching case is depicted in the Figure 9.

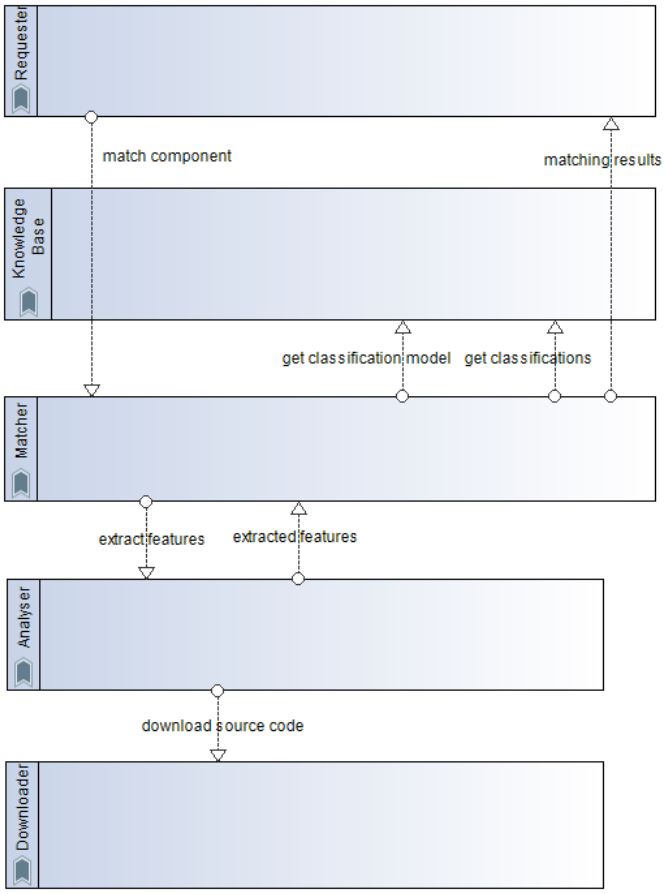


Figure 9 Matching of any arbitrary software component

Profiler Maintainer and CAMEL Modules

The Profiler Maintainer is responsible for constructing the profile of an application as well as maintaining it. In the first case, the component constructs the application profile from the CAMEL model of the application and invokes the NF Model Learner in order to construct an initial performance model for each application component form. In the second case (profile maintenance), it is actually invoked in two situations:

- *CAMEL model update:* it checks whether the update of an application's CAMEL model has led to changing, replacing or adding application components. If this holds, then the application profile will have to be updated. The same process as in the case of profile construction is followed to extract the needed information from the CAMEL model as well as constructing the initial performance models of new components or updating existing models in case of component modifications via the NF Model Learner.
- *New component form verification:* the suggested new forms of a component by the Profiler are validated by the DevOps. Once this validation is finished and the DevOps requires the consideration of the respective new application component form, and it executes through the CAMEL Designer the corresponding method of the Profile Maintainer. In this situation, the Profile Maintainer informs the application profile to highlight that the new component form has been verified. In addition, it requests the NF Model Learner to construct a performance model for this component form. If the method execution succeeds, the CAMEL Designer should update the application's CAMEL model so as to include the new and validated form of the respective application component. This form verification process is depicted in Figure 10 New components form the verification processes.

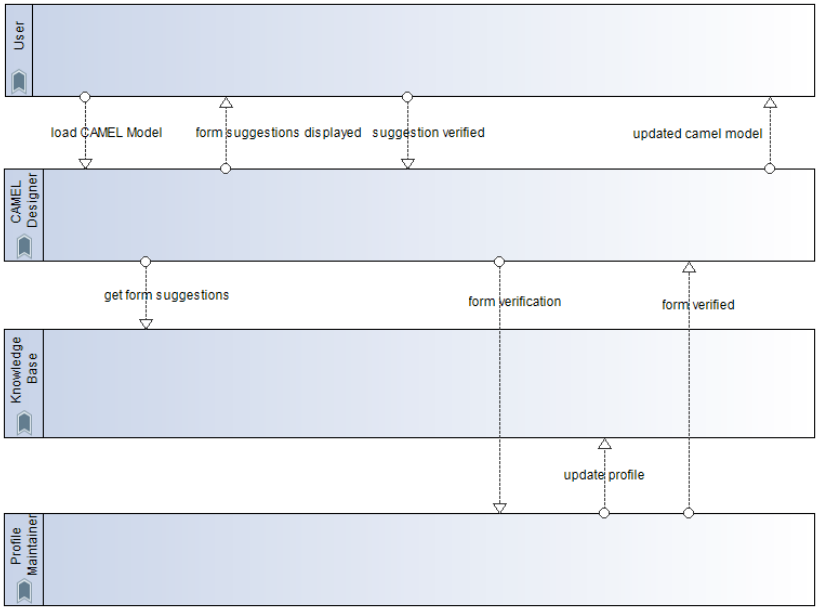


Figure 10 New components form the verification processes

Based on the above analysis, it becomes clear that the Profiler Maintainer can be considered as realising the application profile construction and maintenance responsibility, where the maintenance covers the updating of user application CAMEL models as well as the verification of suggested new application component forms.

3. Code mining components

3.1 Introduction

This chapter is focused on the components of the Application Profiler providing the functionality of code mining. As mentioned in Chapter 1, three components are involved in this functionality: Web Crawler (Section 3.2), Knowledge Base (Section 3.3) and Code Analyser (Section 3.4).

3.2 Web Crawler

Data mining [2] is defined as the identification of information through targeted extrapolation from large single or multiple databases: if needed, data from different sources are crossed to obtain more accurate information. Data crawling techniques [3] (as distributed crawling, general purpose crawling, or focused crawling) are used for data mining.
The techniques and strategies applied to data mining operations are largely automated, consisting of specific software and algorithms suitable for a specific big data [3] in acceptable times, where data are contained in data warehouses scattered around the web and can be heterogeneous and of potentially endless types. They enable us to find associations, anomalies and recurring patterns. The high parallelism and the increasing amount of computing resources available nowadays (alongside highly specialized operators), enables us to apply data mining techniques with efficiency that far exceeds manual analysis. Web Crawlers are used to extract useful sets of information interrelated from data sources on the web. Many types of metadata [4] are provided, such as descriptive, structural, administrative, reference and statistical metadata.
Specifically, in MORPHEMIC, the Web Crawler looks for metadata associated to projects available on known source repositories. At the time of writing this document, a prototype of the Web Crawler has been implemented and running: it is based on the Web Crawler [5] of the MARKOS system. MARKOS is an "*Intelligent Virtual Marketplace for Open Source projects",* the outcome of an EU co-founded research project[5]. One of its Web Crawler's functionalities is to search for open source projects stored in a set of software repositories, analyse the code and the documentation, especially concerning the licenses, and visualize the results. The components produced in the MARKOS project had been designed to be *reusable*. Specifically, they can be separated by the rest of the platform, used as standalone services and re-integrated to other platforms regardless of the implementation details. The licence, Mozilla Public Licence 2.0[6], allows to reuse the Web Crawler in the MORPHEMIC project and to modify it accordingly. Specifically, the MARKOS Web Crawler can be used as a standalone service to provide metadata of open source projects and to get references for the source code.
The MARKOS project was focused on software licensing, so the MARKOS' Web Crawler aims at this objective. However, most of the metadata retrieved are useful for software categorization as well: for this reason, the MORPHEMIC's Web Crawler can reuse most of the functionalities provided by the MARKOS Web Crawler. For the current release of MORPHEMIC, some preliminary modifications have been introduced. Others are already planned and it is possible that the results of the project will suggest more in the future. So far, one main change concerns the interaction with GitHub and Apache repositories. The current version of the MARKOS' Web Crawler does not interact directly with them, but uses Flossmole[7] (a collaborative collection of Free, Libre, and Open Source Software (FLOSS) data) as a mediator. However, Flossmole has not updated GitHub and Apache projects since 2017, so it is not currently useful for MORPHEMIC. As a consequence, Flossmole has been removed from the list of the configured source code repositories and a module to directly interact with GitHub (Section 4.3.1) and Apache (Section 4.3.2) has been implemented and integrated. A second important change with respect to the MARKOS' version concerns the components aimed at retrieving metadata. Specifically, an analysis on the attributes and metadata needed by MORPHEMIC has been performed. This analysis aimed at determining whether all the needed metadata were provided by the main repositories and whether any formal modification was needed.
Figure 11 shows the architecture of the MARKOS Web Crawler. All its components have been inherited by MORPHEMIC but the DOAP schema, which has been included in the Knowledge Base (Section 3.3).

[5] https://cordis.europa.eu/project/id/317743
[6] https://www.mozilla.org/en-US/MPL/2.0/
[7] https://flossmole.org/

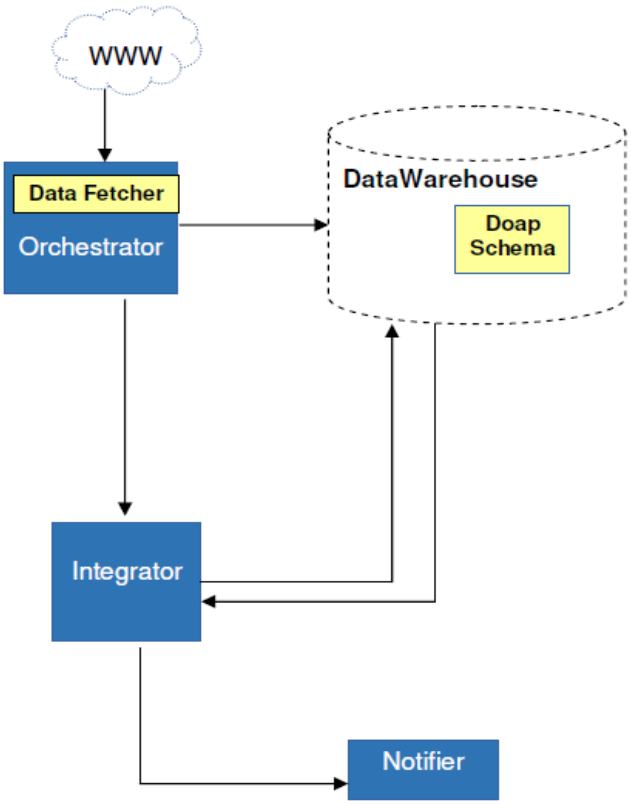


Figure 11 Architecture of MARKOS's Web Crawler

The DOAP schema stores data and metadata of the retrieved open source projects at the end of the crawling process. Such data and metadata are stored in a unique standard format regardless their origin: specifically, some repositories already provide data in DOAP standard format (for example Apache projects, Section 4.3). In the other cases, specific conversion mechanisms have been implemented starting from the format in which data are provided (zip page, Json, XML or HTML, Section 4.3).
In the second release of MORPHEMIC, DOAP support will be part of the Knowledge Base (more details are provided in Section 3.3).

3.2.1 Architecture of the Web Crawler

The architecture of the MORPHEMIC's Web Crawler is shown in the following figure:

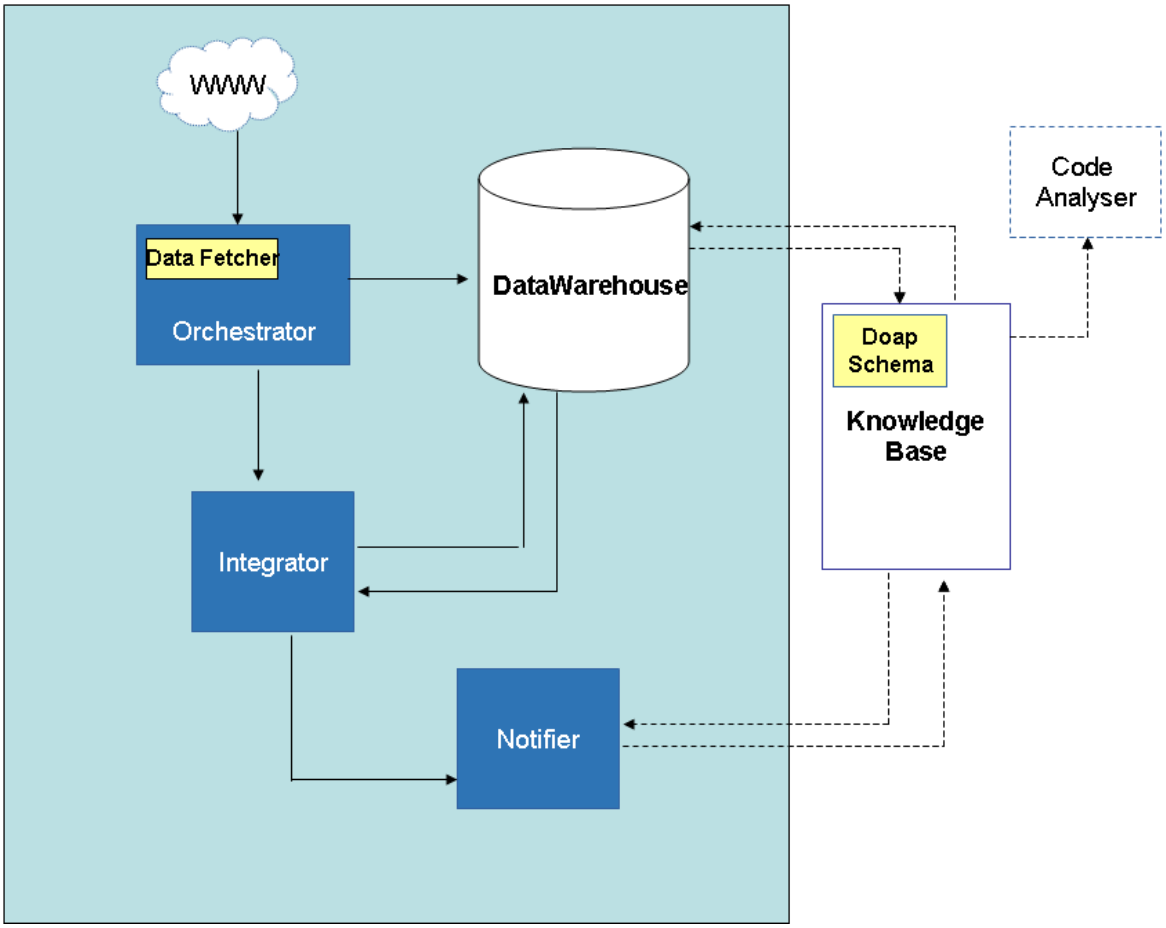


Figure 12 Architecture of MORPHEMIC's Web Crawler

The components of MORPHEMIC's Web Crawler are following:

- *Orchestrator:* it contains the Data Fetchers and coordinates their work. In particular data fetchers are clients dedicated to specific source code repositories (e.g., GitHub, Apache). The details about how the orchestrator and the data fetchers work are provided in the section 3.2.2, "Orchestrator" and "Data Fetcher".
- *Data Warehouse:* stores the data downloaded from the repositories, including data of the DOAP. In the first release, the DOAP tables will be part of the data warehouse, starting from the second release, just they will be included in the Knowledge Base**.**
- *Integrator:* it is responsible for grouping the similar pieces of information coming from different sources; more details are provided in the Section 3.2.2 "Integrator".
- *Notifier*: searches for batches of integrated projects that have been changed during the last iteration. Specifically, a project is flagged as changed if:
  - metadata has been changed with respect to the previous version;
  - there is a new release or information about an existing release has changed.

  If a project has *changed*, the Notifier sends a notification to the Code Analyser, through the Knowledge Base, identifying the changed project.

3.2.2 Functionalities and process flows

The functionalities provided by the Web Crawler are the following:

- *maintain* a list of data sources and structured information related to open source projects where:
  - forges are accessed directly, via API or crawling on the site
  - services offer data aggregated from different forges
- *download* automatically and continuously part of that information as raw data

- *prepare* an integrated structure that stores each information token together with its own source when information about a specific project can be retrieved from different sources
- *interact* with the Knowledge Base to store the information downloaded and parsed
- *make available* the information processed as a running RESTful API service
- *notify* the Code Analyser when new releases have been made available;
- *continuously crawl* the repositories/websites of all projects looking for updated releases.

The interactions of the crawling process are shown in Figure 13. The process gets data from external repositories, through the web, and stores them on a specific storage, as mentioned above, part of the Knowledge Base.

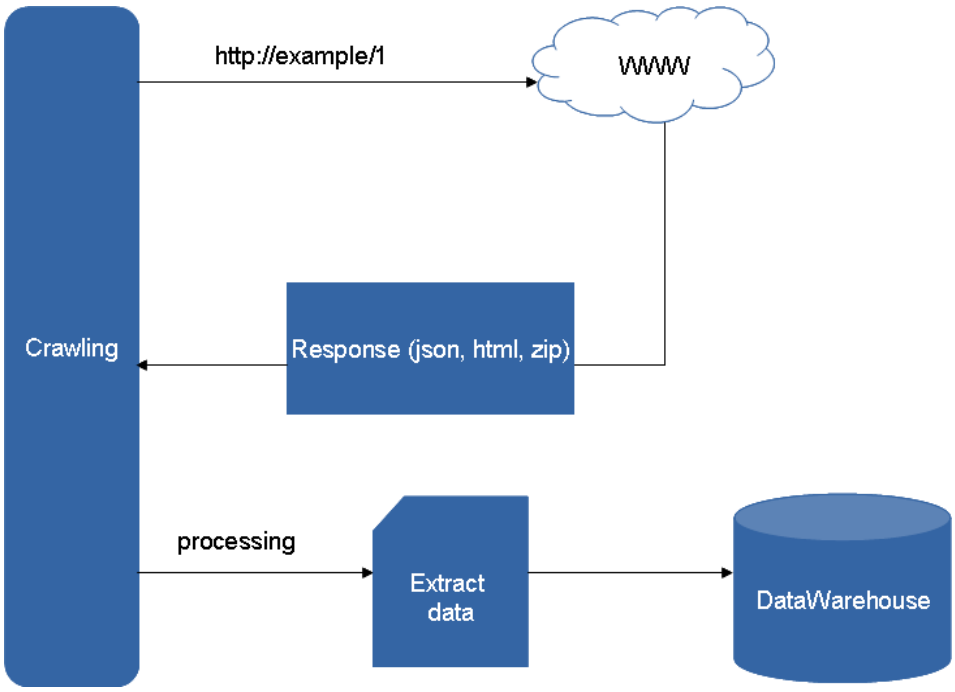


Figure 13 Interactions of the crawling process

Data Warehouse

The Data Warehouse, as mentioned in the previous section, stores data retrieved from the repositories.
Specifically, the Data Warehouse is an internal data store in the Web Crawler and it can be exploited from the Knowledge Base. It is implemented as a set of tables on a relational database (i.e., MySQL[8] and MariaDB[9]), containing the following information:

- *raw data*, i.e. not-processed information downloaded from the repositories. These tables containing these data are identified with the prefix "RAW" followed by the name of the specific repository: for example, RAW_Apache, RAW_GitHub, RAW_Jquery;
- *DOAP data*, i.e. data describing the retrieved projects. These data are shared with the Code Analyser. All these tables are compliant with the standard DOAP RDF schema[10]. The DOAP tables are identified with the prefix "DOAP" followed by the name of the specific piece of information provided as defined in the DOAP RDF schema. For example, the table called DOAP_Project provides refers to the fetched project (such as name of the project, description, category, home page,etc). The table DOAP_Repository provides information about the repository used by the fetched project (i.e.location of the repository).

[8] https://www.mysql.com/
[9] https://mariadb.org/
[10] https://www.w3.org/wiki/SemanticWebDOAPBulletinBoard

Data Fetcher

Data fetchers are sub-components of the Orchestrator acting as clients and implemented according to the specific features of the associated source code repositories. It is possible to implement a specific data fetcher if a new source code repository is associated. At the time of writing this Deliverable, the implemented Data Fetcher (in the MARKOS's Web Crawler) up and running are:

- *GitHub* (Section 4.3.1);
- *Apache* (Section 4.3.2);
- *jQuery Plugin* (Section 4.3.3).

As mentioned before, one of the most important modifications of MORPHEMIC's version with respect to the MARKOS' one concerns the metadata retrieval process for GitHub. Specifically, the Data Fetcher for GitHub in MARKOS was based on Flossmole[11] while in MORPHEMIC it is directly implemented. In general, the details of the procedure by which each data fetcher interacts with the associated source code repository depend on the respective implementation and configuration. These details include the timing (e.g., a certain data fetcher may actually contact the source code repository on a daily basis or on a regular basis every 7 or 15 days) and the data format (for instance, the Apache Software Foundation provides a web page containing a list of the URLs of all DOAP[12] files for the projects it hosts, while other repositories provide specific web services) as reported in the dedicated section, Orchestrator.

Orchestrator

The Orchestrator defines the scheduling of the polling processes, and, in general, the timing of the whole crawling process. Specifically, it continuously polls each associated source repository looking for new data or metadata. The polling time is configurable (e.g., once a day) and the interaction with a certain source code repository happens through a specific Data Fetcher.
As mentioned above, the polling period is configurable: actually, the configuration file of the Web Crawler contains several pieces of information, some of which are *generic,* others are *per-fetcher*. The following list is part of the configuration file and contains some of the configuration fields. For example, the general *reporitory_crawler_sleep_time* defines the polling time while *apache_every_n_days* defines how frequently (how many days) the Apache data fetcher should retrieve the relevant data:

```
[General]
 sleep_time=30000
 notifier_sleep_time=60
 repository_crawler_sleep_time=30060
sf_file_path=/home/people/markos/markos02/data-for-doap-work
 flossmole_file_path=/home/people/markos/markos02/flossmole
 temporary_directory=/tmp
 exit_now=False

[Fetchers]
 # a negative number (e.g. -1) disables the source
 apache_every_n_days=-1
 github_every_n_days=1
 jqueryplugin_every_n_days=-1

[RepositoryCrawler]
  github_archive_months=3
 # how many events make a project interesting for the code analysis
```

[11] https://flossmole.org/
[12] https://www.w3.org/wiki/SemanticWebDOAPBulletinBoard

github_top_projects_event_count=100
……
……
……

The sequence diagram explicating how the generic data fetcher works is shown in Figure 14. It includes data fetching tasks and persistence in the Knowledge Base (a generic Datawarehouse in the diagram), too.

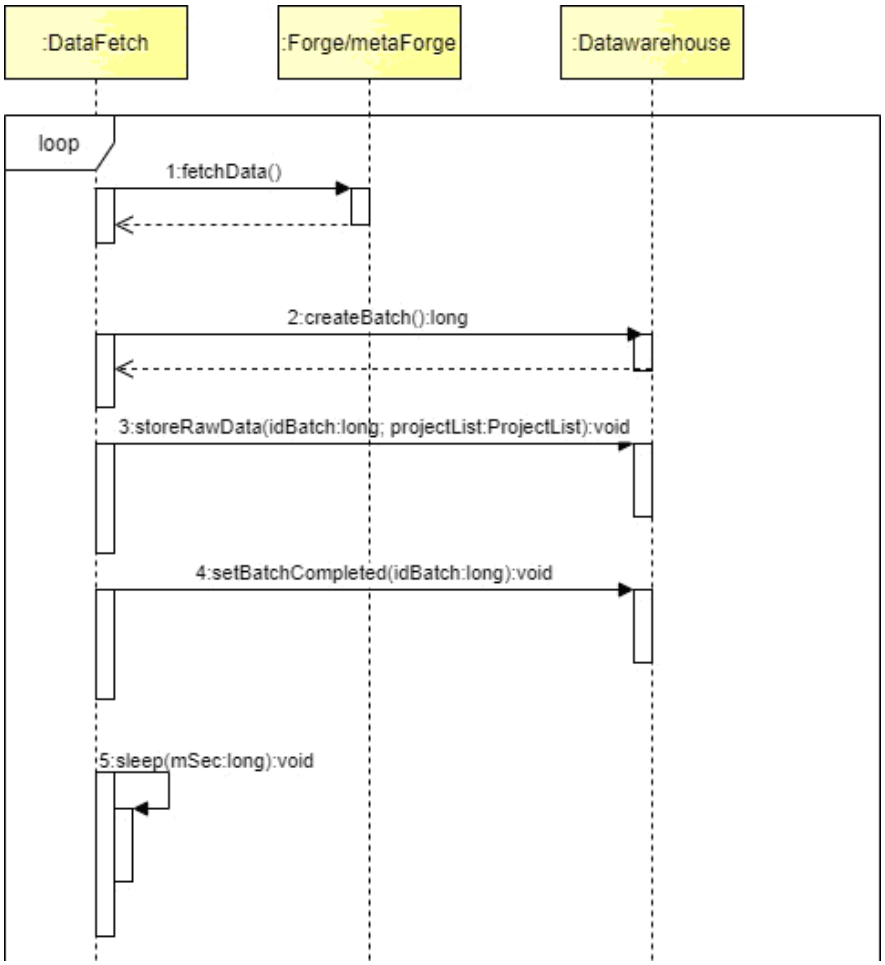


Figure 14 How a data fetcher works

Integrator

After the fetching process is completed, the downloaded data are stored in general locally in specific batch files.
At this point the Integrator checks if instances of the same project are stored on different source code repositories. In order to achieve this objective, the Integrator extracts all the associated projects records from the downloaded batch files and tries to match the projects according to some key elements, such as name homepage, repository URL, etc. If the matching process results in projects that are different instances of the same project, the Integrator aggregates their data and stores them. It is possible to summarize the integration process in the following steps:

- *get* each information and store it along with its source;
- *extract* all the projects' records associated with the unique identifier (referred to as batch) and, for each of them, find existing projects matching the name of the project;
- *all the data* coming from different sources are aggregated and stored on dedicated tables (called integrated data table).

Figure 15 shows the sequence diagrams of the integration process, which includes the notification to the Code Analyser.

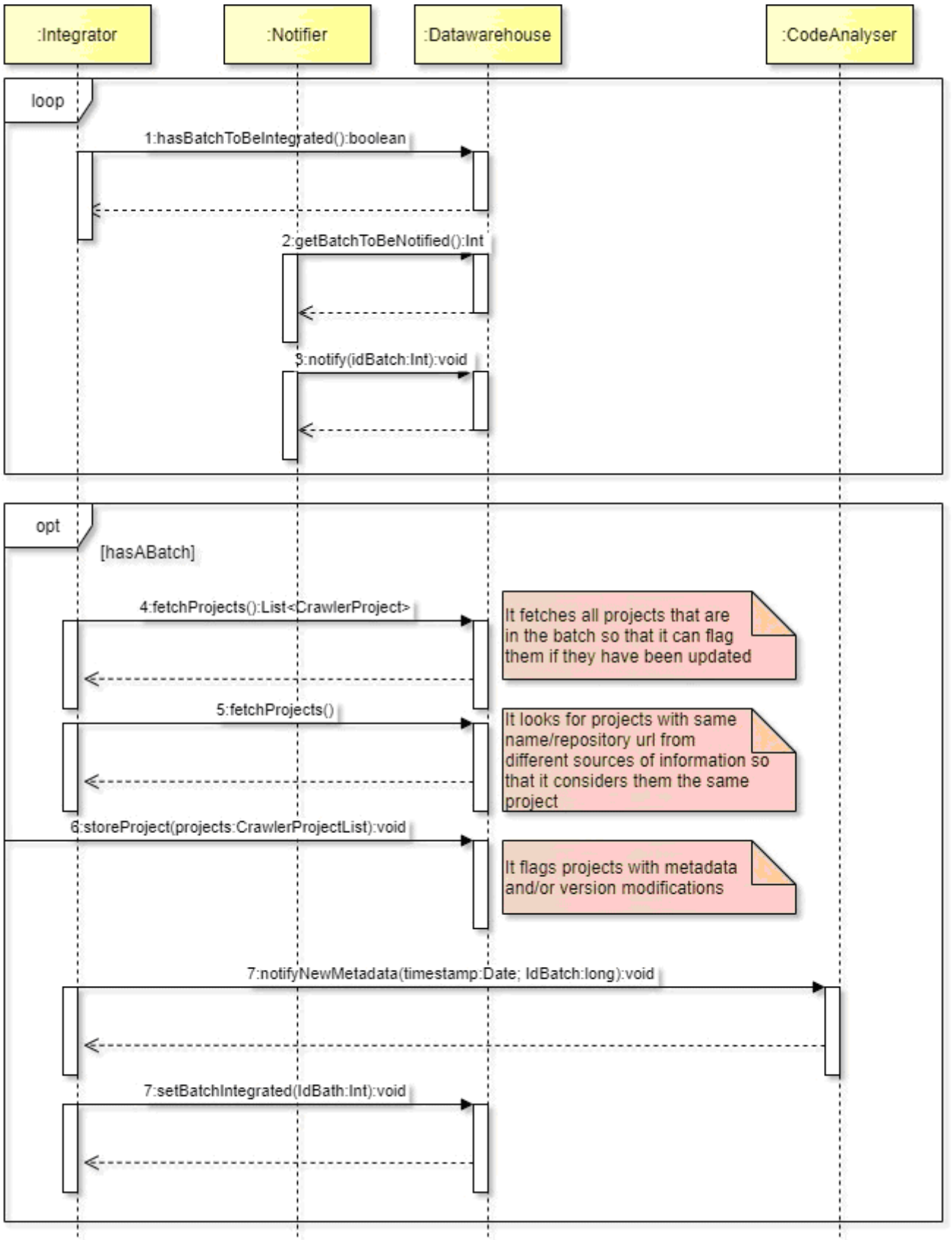


Figure 15 Sequence diagram of the integration process

Notifier

At the end of the process, the Web Crawler will provide the Knowledge Base with integrated data from all the sources referring to the same project. The project name is the key that can be used to search for this information.
A notification is also sent to the Code Analyser to inform you that a new integrated project has been stored.
The notification component notifies the Code Analyser in case of *change.* Each project during the integration phase can be flagged as changed if:

- *new release or new information* about existent release has occurred;
- *metadata information* changes with respect to previous change.

A notification includes the identifier of the project's batch to which it is referred. If required, other components of the Application Profiler (such as the Analyser) will retrieve the entire project in a batch and process it synchronously.
The Web Crawler fetches information about open source projects from a heterogeneous set of sources (open source software forges and others are meta-forges). The process is very repository-specific, since each forge, Meta forge, list of directory projects, list of packages provides data and metadata in a different way. The MARKOS version already included some data fetchers to get data from different repositories. In general, it is possible to implement other data fetchers in case during the MORPHEMIC project other repositories might be considered relevant as well.

3.2.3 Hardware and Software Requirements

The Web Crawler is released under the Mozilla Public License, v. 2.0 and is written in Python2.7. It can be deployed on Ubuntu 18.04 LTS and needs to be associated with a relational database to store the metadata. It has been tested with MySQL[13] and MariaDB[14].
The minimum hardware requirements to install the Web Crawler are:

- 2 cores CPU;
- 4 GBs of RAM;
- 2 GBs of disk space per month's information.

An implementation of this web crawler is available for testing purposes[15]

3.3 The Knowledge Base

The Knowledge Base is a repository responsible for managing (storage, update, delete, searching) all the knowledge collected for the user application and its profile as well as the open source software components crawled. Furthermore, the Knowledge Base plays the role of an intermediate communication medium among the components of the Application Profiler. For example, the Web Crawler uses the Knowledge Base to store the metadata about the crawled projects. The Knowledge Base is then acceded by the Classifier to exploit those metadata in order to realise its classification functionality. Finally, it should be noted that the Knowledge Base can interface with the CAMEL Designer (as showed in Figure 4) in order to enable users to inspect the functional matches of their application components. Such an inspection can then facilitate users in verifying new application component configurations through the use of the Profile Maintainer. This can finally lead to updating also the application's CAMEL model with the verified component configurations.
As mentioned above (Section 3.2.2), the DOAP Schema, part of the architecture of the MARKOS' Web Crawler, starting from the second review of MORPHEMIC, will be included in the Knowledge Base. Specifically, it will integrate the data storage functionality that will be acceded through RESTful web services by all the components of the Application Profiler. The following figure shows the details:

[13] https://www.mysql.com/
[14] https://mariadb.org/
[15] https://www.fiware.org/

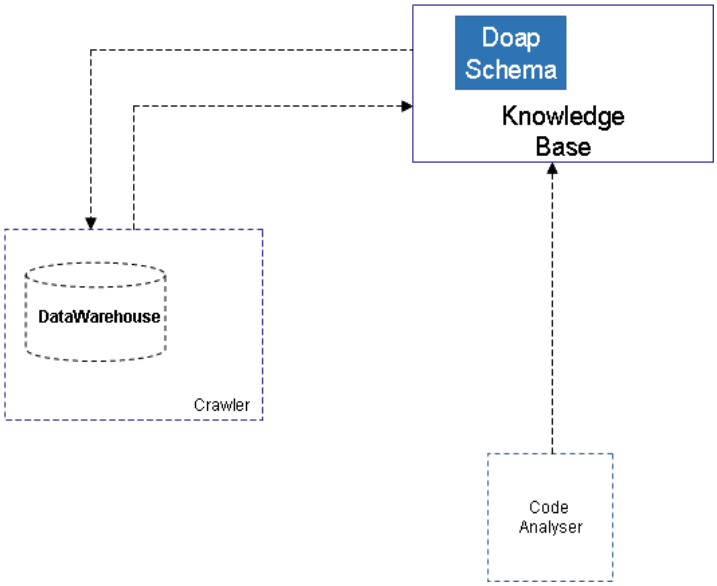


Figure 16 Overview Architecture of Knowledge Base

3.4 Code Analyser

The Code Analyser processes the source code downloaded from the repositories. It has two main responsibilities:

- *retrieval* of the metadata related to quality and security;
- *extraction* of functional features.

Figure 17 represents the collaboration diagram describing the interactions that are involved in the process of code analysis concerning the interaction with the related components of the Application Profiler.

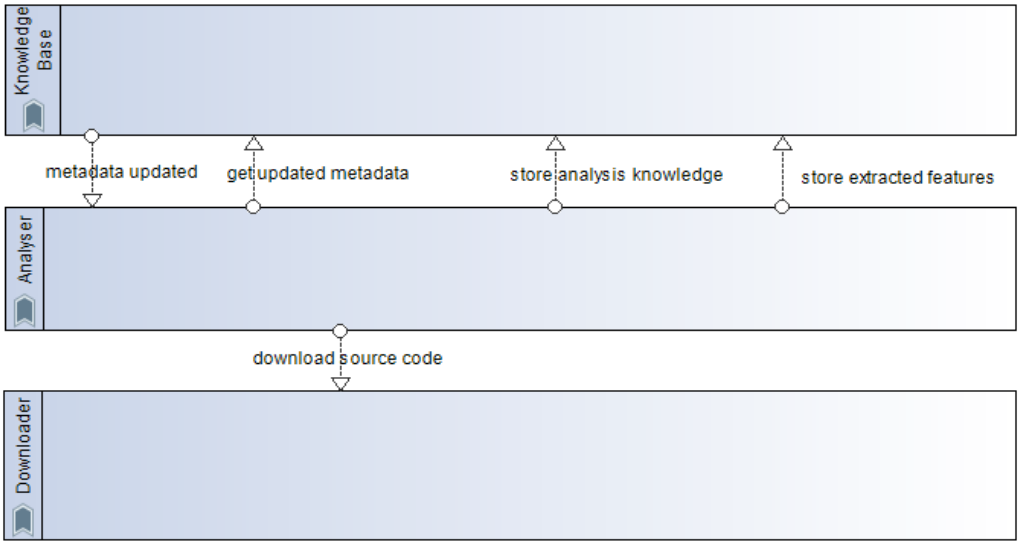


Figure 17 Analysis functionality fulfilment through a collaboration diagram

The first responsibility for this component is to check the quality and security level of the source code of crawled software components through the use of appropriate techniques, such as *static analysis* (see Section 5.1.4*)*. Its main goal is to produce additional knowledge (for crawled software components), to be stored in the Knowledge Base, that can assist in the ranking of functional matching results (between application and crawled software components). The main rationale is that DevOps should be assisted in selecting the right software component from the ones matched based on the software component quality and security level. As a result, through the right component selection, the overall application quality and security level is maintained or even enhanced.
Another responsibility of this component is to conduct feature extraction, like for example, to produce the right input that is then amenable for (functional) classification. The feature extraction can result in features represented in

different forms. For instance, a bag or vector of words can be one form while a control flow or program dependence graph another. Irrespectively of the form, the extracted features are imported back to the Knowledge Base to further enrich the metadata specification of open source software components.
It must be highlighted that the quality, security and functional feature knowledge is not only produced the first time a certain component is being processed: if the source code changes the knowledge will be updated, re-executing the crawler process. Thus, this knowledge is not only produced but also properly maintained.
Finally, it must be noted that this component pre-supposes that the Downloader has already downloaded the source code that has to be analysed.

4. Evaluation of the Crawling process

4.1 Introduction

As mentioned above, the code mining components search for data on the web. Specifically, the Web Crawler is configured with pointers to a set of open source project repositories from which metadata and source code can be retrieved.
This chapter will provide an evaluation of the current implementation of the Web Crawler (Section 3.2) and of the associated and associable repositories (Section 4.3).

4.2 Evaluation of the Web Crawler

The starting point of MORPHEMIC's Web Crawler is MARKOS' Web Crawler. The choice has been defined following an evaluation process in which some alternatives have been compared. This section provides the results of this comparison, in particular the following tools have been considered:

- MARKOS[16];
- OpenHub[17];
- Krugle[18];
- ScanCode;[19]
- SearchCode[20].

The parameters used to evaluate these tools are:

- *Description* of the project.
- *Operating mechanism:* how the tool works and its main functionalities.
- *URL:* reference link to the website (if exist).
- *Date of creation:* to understand if the project is new or consolidated.
- *Status:* if the tool is currently used by at least one community or is deprecated.
- *Support:* if any kind of support is provided.
- *Forum:* if a forum exists to share information with a community.
- *Documentation:* availability of the documentation (API, source code and/or download link).
- *Ownership:* who is the owner?
- *License:* in particular if the tool is an open source or, at least, freely usable.
- *Database:* how the information is stored, how the collection data are managed and analysed.
- *Programming language:* the programming language used to implement the tool.

This information is reported for each tool in the following tables (from Table 1 to Table 5).

[16] https://cordis.europa.eu/project/id/317743
[17] https://www.openhub.net/
[18] https://krugle.webtuits.com/
[19] https://github.com/nexB/scancode-toolkit
[20] https://searchcode.com/

Table 1 MARKOS Web Crawler

MARKOS	
Description of the project	MARKOS Web Crawler provides an architectural and semantic classification of the retrieved projects. MARKOS inspects the code structure showing the components, their internal and external dependencies and their interfaces.
Operating mechanism	MARKOS Web Crawler is a RESTful web service. The Web Crawler retrieves the structured or unstructured content of open source projects. The analysis tools retrieve the information from the description and license of the analysed software. The produced information is stored in a relation database (MySQL or MariaDB) for the publication of linked data and for supporting queries from the front-end side. The Web Crawler visits the registered project sites and notifies the existence of new or modified open source projects using specific communication protocols.
URL	https://sourceforge.net/projects/markosproject/files/MARKOS%202.0/sources/
Date of creation	2013
Status	Active
Support	Yes
Forum	No
Documentation	ENG is the owner of the Web Crawler and can provide the documentation.
Ownership	Engineering Ingegneria Informatica
License	Mozilla Public License, v. 2.0
Open Source	Yes
Database	MySQL or MariaDB
Programming Language	Python

Table 2 OpenHub Web Crawler

OpenHub	
Description of the project	OpenHub (or Ohloh) is an open source directory that anyone can access. It provides statistics on projects, their licenses (including possible conflicts between various licenses) and software metrics (such as number of lines of code or commit statistics for new code releases, in particular). It features comprehensive metrics and analytics on thousands of open source projects.
Operating mechanism	OpenHub is a Restful web service and gets the needed information from versioning and revision systems (such as CSV, SVN or GIT).
URL	www.openhub.net
Date of creation	2006
Status	Active
Support	Yes
Forum	Yes
Documentation	Yes
Ownership	Black Duck Software
License	Proprietary
Database	PostgreSQL
Programming Language	Ruby

Table 3 Krugle Web Crawler

Krugle	
Description of the project	Krugle is a search engine that enables to locate open source code and quickly share it with other developers. Krugle provides mainly statistics (number of commits per user, evaluation of the competences of the developers working on a certain project)
Operating mechanism	Krugle collects through modern crawling and research technologies specifications, project plans, defect tracking records, build records and source code. This information is organized using the rich metadata captured by these systems and loaded into dedicated databases. The information is accessed via REST services.
URL	https://www.krugle.com/
Date of creation	2009
Status	Active
Support	Yes
Forum	Yes
Documentation	Yes
Ownership	Aragon Consulting Group
License	Enterprise Edition is a Proprietary license. However, in the engine's website, there is a engine demonstration version: https://opensearch.krugle.org
Database	Not available
Programming Language	Not available

Table 4 ScanCode Web Crawler

ScanCode	
Description of the project	ScanCode detects licenses, copyrights, manifests, packages and more by scanning the code. ScanCode is a monitoring tool, a License Management and a Web Application.
Operating mechanism	The tool works locally on the code stored on a specific machine, so it needs to be installed. Specifically, ScanCode collects all the information useful for license analysis (and more) in a database and allows : 1) to collect an inventory of code files and classify code using file types; 2) to extract files from any archive using a generic extractor, able to extract texts from binary files; as well 3) to use an extensible rules engine to detect open source license text and communications; 4) to use a specialized parser to capture copyright statements; 5) to identify the package code and collect metadata from packages; 6) to report the results in different formats (JSON, CSV, etc.) to be used with other tools or with the browser.
URL	https://github.com/nexB/scancode-toolkit
Date of creation	Not available
Status	Active
Support tool	Yes
Forum	Yes
Documentation	Yes
Ownership	nexB
License	Apache-2.0 with an acknowledgment required to accompany the scan output. Public domain CC-0 for reference datasets. Multiple licenses (GPL2/3, LGPL, MIT, BSD, etc.) for third-party components.
Database	Not available
Programming Language	Python 2.7

Table 5 SearchCode Web Crawler

SearchCode	
Description of the project	SearchCode is a free source code and documentation search engine. The API documentation, code snippets, and the open source repository (free software) are indexed and searchable. It has indexed several billion lines of code and more than 90 languages.
Operating mechanism	Indexes and makes searchable code snippets and open source (free software) repositories available on the web.
URL	https://searchcode.com/
Date of creation	2014
Status	Active
Support tool	Yes
Forum	Yes
Documentation	Yes
Ownership	SearchCode
License	The starter version offers unlimited users, identifies more than 100 languages, secure APIs. It is possible to perform basic operations (e.g. search) but APIs can only be modified for a fee : https://searchcodeserver.com/pricing.html. The other versions are paid as reported by the link above.
Database	MySQL
Programming Language	Java

The following table summarizes the information on the analysed tools:

Table 6 Summary of the analysis of the code mining tool

	MARKOS	OpenHub	Krugle	ScanCode	SearchCode
Documentation	Yes	Yes	Yes	Yes	Yes
License	Open Source	Proprietary software.	Proprietary Software	Open Source	Starter version is free, but the operations cannot be modified and extend the API: https://searchcodeserver.com/pricing.html
Database	MySQL, MariaDB	PostgreSQL	N/A	N/A	MySQL
Programming Language	Python, Java	Ruby	N/A	Python	Java
Status	Active	Active	Yes	Active	Active
Support Tools	Yes	Yes	Yes	Yes	Yes
Forum	No	Yes	Yes	Yes	Yes
Brief description of the search and crawl processes	Focus on the software structure	Focus on the project structure	Focus on the offer's user statistics	Focus on the monitoring tool	Focus on source code and search engine documentation

MARKOS was selected due to the fact that:

- *it focuses on the software structure* (e.g., on Dependencies between components, interfaces etc.) and not on the structure of the project (e.g., dependencies between activities and people as for example OpenHub);
- its components can be re-used and extended for the MORPHEMIC platform;
- *no limitation* for the license of use (as in the case of SearchCode) since it is open source (and not proprietary like, for example, Krugle or OpenHub);
- *the database is available and known.* For some of the analysed tools the used database is not clear (as for example ScanCode or Krugle).

MARKOS provides RESTful APIs. The advantages of this technology are:

- *independence*: independent from languages and platforms;

- *client-server separation*: allows us to independently deal with the evolution of the various components and the interface can be used on different types of platforms;
- *scalability*: if the platform scales the interfaces do not change.

Unlike other tools, MARKOS retrieves the content of open source projects and makes a deep inspection of the code structure showing the components, their internal and external dependencies, and their interfaces (not just the number of developers, release number, commits).
The MARKOS system mainly analyses the software and the metadata contained in the repositories that offer public and standard login interfaces (e.g., CVS, SVN, GIT) by using well-known protocols (e.g., Linked Data using the DOAP).
The components produced in the MARKOS project are designed to be reusable. Specifically, they can be separated by the rest of the platform, used as standalone services and re-integrated in other platforms regardless of the implementation details.
In particular, the MARKOS Web Crawler can be used as a standalone service to provide metadata of open source projects usable for MORPHEMIC. The MARKOS Web Crawler will be easily integrated in the MORPHEMIC platform through the Restful APIs. Some modifications were implemented or are planned (section 3.2), mainly to interact with other source code repositories or to make integration easier at architecture level. However, the main functionalities will be preserved.

4.3 Evaluation of the open source project repositories

The projects and software data are actually distributed on several existing forges and meta-forges on the Web. The Web Crawler retrieves the projects metadata, compliant to the description of a project format, which provides all information about the application project processed and contains information like project name, description, URI of source code repository, and so on. The data gathered by the Web Crawler are stored in the Knowledge Base.
Every repository supports a different way to collect the project information: API, SQL format, HTML, json, xml or compressed file.
For the MARKOS project, the selection of the source code repositories to be associated was performed based on a survey among the technical partners, who had been requested to evaluate their interest on a list of meta-forges (e.g., Ohloh, Flossmole) and forges (e.g., GitHub). The same approach has been used in MORPHEMIC: specifically, a set of active repositories has been presented to the technical partners. The result of this work is documented in sections from 4.3.1 to 4.3.3. Their evaluation is based on the following information:

- *where* the data is taken from;
- *how* the downloaded data looks like (zip, xml, json, etc.);
- *what* is the information parsed, loaded and integrated;
- *what* is the complete set of information produced by each repository.

Another activity has been studying and analysing other possible repositories that could be useful, even if not already associated for the crawling to the MORPHEMIC platform. Section 4.3.4 provides a description of the work conducted for this purpose.

4.3.1 GitHub

GitHub[21] is a web and cloud-based service that helps developers to store and manage their code and track & control changes. The information provided highly depends on the forge it comes from; the information about GitHub is provided through the GHArchive[22]-which collects information on the *events* of the projects. More than twenty types of events are provided as new commits, fork events, opening new tickets, commenting, and adding members to a project. Events are collected into hourly archives: each of them contains JSON encoded events which can be acceded[23] with any HTTP client.

[21] https://github.com/
[22] https://www.gharchive.org
[23] https://docs.github.com/en/free-pro-team@latest/rest

As reported in the GHArchive documentation[24], the only recommendation is to restrict the queries to relevant time ranges to minimize the amount of the data considered. Specifically, the processing of up to 1 TB of data per month is free of charge.

Use of MORPHEMIC

GitHub is an important source of information for MORPHEMIC as it aggregates data from a plethora of different heterogeneous projects.
The Web Crawler accedes directly GHArchive that provides the information of the events associated to each project in the XML format. The project list is retrieved by inspecting the events associated to the projects. Moreover, the events contain most of the information needed by MORPHEMIC. The list is shown in Table 7 Example of the data that can be provided by GHArchive.

Table 7 Example of the data that can be provided by GHArchive

Field	Description	Example
Id	Unique identified for the event	*"13087388045"*
Type	The type of the event	*"Release Event"*
Actor	Actor generating event. It is provided in a Json format	*{"id":41898282,"login":"github-actions[bot]","display_login":"github-actions","gravatar_id":"", "url":"https://api.github.com/users/github-actions[bot]", "avatar_url":"https://avatars.githubusercontent.com/u/41898282?"*.....
Repo	The repository is associated with the event. It's provided in a Json format.	*{"id":284021818, "name":"grische/blender", "url":"https://api.github.com/repos/grische/blender"*.....
Payload	Payload depending on the release Event Type. It can provide information about the release. It is encoded in Json format.	*{"tarball_url":"https://api.github.com/repos/grische/blender/tarball/v2.90.0-72b422c1e101","zipball_url":"https://api.github.com/repos/grische/blender/zipball/v2.90.0-72b422c1e101","body":""*
Created	Timestamp of associated event	*"2020-08-02T11:59:48Z"*
Public	Type of Boolean: true if the event is public, false otherwise.	*True*

4.3.2 Apache

Apache is the source code repository developed by the Apache Software Foundation[25].
In order to be published, the projects must be approved by the Apache Project Management Committee (PMC). After that, an approved project is added in an XML that links to individual DOAP files[26]:

[24] https://www.gharchive.org/
[25] https://www.apache.org/
[26] https://svn.apache.org/repos/asf/comdev/projects.apache.org/trunk/data/projects.xml

```
<!-- Licensed to the Apache Software Foundation (ASF) ..... See the License for the specific language governing permissions and limitations under the License. -->
<!-- Project DOAP files ================== Each PMC (committee) may manage one or more projects, each of which should have a DOAP listed here. This list may include projects that have been retired. The PMC descriptor files are listed in the file committees.xml (in this directory) -->

<doapFiles>
<!-- was in projects-old https://svn.apache.org/repos/asf/infrastructure/site-tools/trunk/projects/files.xml -->
<!-- This file is ordered by committee and then by project managed by corresponding PMC -->
<location>http://svn.apache.org/repos/asf/abdera/java/trunk/doap_Abdera.rdf</location>
<location>https://accumulo.apache.org/doap/accumulo.rdf</location>
<location>http://svn.apache.org/repos/asf/ace/doap.rdf</location>
<location>http://svn.apache.org/repos/asf/activemq/trunk/doap.rdf</location>
<location>https://svn.apache.org/repos/asf/airavata/airavata_doap.rdf</location>
<location>https://svn.apache.org/repos/asf/allura/doap_Allura.rdf</location>
.....
<location>http://zookeeper.apache.org/doap.rdf</location>
<!-- This file is ordered by committee and then by project managed by corresponding PMC -->
</doapFiles>
```

For each project list, the Web Crawler downloads an xml file that provides the information. Each reference represents a project. The crawler points in a loop to each reference, and fetches the xml file that contains the information of the selected project.

Use of MORPHEMIC

The information of the projects retrieved by the Web Crawler is provided in a DOAP XML standard format. Currently, the following information is provided:

- *Name* of the project.
- *Description* of the project.
- *Programming language* (e.g., Java, Python, etc).
- *Release version:* list of the versions released for the selected project.
- *Download page:* reference link where it's possible to download the source code of the selected project.
- *Date:* release version date.
- *Maintainer:* name of the user that maintains the project (one or more users).
- *License:* type of License (e.g., Apache 2.0).
- *Homepage:* reference link (e.g., http://brooklyn.apache.org).
- *Created:* date of creation of the project (e.g., 2016-02-23).

Additional information could be provided if needed as reported in Table 8 Example of data that can be provided by one Apache project:

Table 8 Example of data that can be provided by one Apache project

Field	Description	Example
Name	Name of the project	*Apache Brooklyn*
Description	Full description of the project	*Brooklyn is about deploying.... and managing applications: composing a full stack for an application; deploying to cloud and non-cloud targets; using monitoring tools to collect key health/performance metrics; responding to situations such as a failing node; and adding or removing capacity to match demand*
Created	Date of project creation	*2016-02-23*
License	Type of license	*Apache-2.0*
Homepage	Reference Link	*http://brooklyn.apache.org*
Short description	Short description of the project	*Apache Brooklyn is a framework for modelling, monitoring, and managing applications through autonomic blueprints*
Bug database	Reference link to the issue home page project	*https://issues.apache.org/jira/browse/BROOKLYN/*
Mailing-list	Reference link to the community	*http://brooklyn.apache.org/community/mailing-lists.html*
Download page	URI of the download page	*http://brooklyn.apache.org/download/index.html*
Programming language	Programming language used for the project	*Java*
Category	The name of the category to which the project belongs to	*Cloud*
Release	List of versions for that release. For each release other information is provided: name, date of creation and revision.	*<release>* *<Version><name> 0.11.0</name><create>2017-05-18</create>* *<revision>0.11.0</revision></Version>* *</release>* *<release>* *<Version><name> 0.12.0</name><create>2017-12-18</create>* *<revision>0.12.0</revision>* *</Version>* *</release>* *.....*
Repository	List of repositories. For each repository other information such as the name of the repository, location and browser are provided.	*<repository>* *<GitRepository><location rdf:resource="https://git-wipus.apache.org/repos/asf/brooklin.git"/<browser rdf:resource="https://git-wip-us.apache.org/repos/asf?p=brooklyn.git"/></GitRepository>* *</repository>* *.....*
maintainer	List of the users that maintain the project	*<maintainer><foaf:Person><foaf:name>MarioRossi* *</foaf:name><foaf:mbox* *rdf:resource="mailto:mariorossi@apache.org"/</foaf:Person* *</maintainer><maintainer><foaf:Person> <foaf:name>John Brambilla</foaf:name> <foaf:mbox rdf:resource="mailto:jbramb@apache.org"/></foaf:Person> </maintainer>* *.....*

4.3.3 jQuery Plugin Registry

jQuery Plugin Registry[27] is a JavaScript library for web applications, distributed as free software, under the terms of the MIT License. The goal is to simplify the selection, manipulation, event handling and animation of DOM elements in HTML pages, as well as simplify the use of AJAX functionality. jQuery Plugin Registry is a site containing a list of the available plugins by indexing some projects included in a set of GitHub's repositories.

[27] https://plugins.jquery.com/

Use of MORPHEMIC

MOPRHEMIC will use the list of the projects provided in HTML format. Specifically, the Web Crawler parses the list, gets the information and stores them in the Knowledge Base. The information provided is the following:

- *Attribution:* name of the owner or developer (ex. Jack Moore).
- *Entry-title:* name of the plugin (ex. ColorBox).
- *Description:* description of plugin (ex. 'jQuery lightbox and modal window plugin').
- *Download link:* where the user can download the plugin (e.g., http://github.com/jackmoore/colorbox/zipball/1.5.14).
- *Name:* the name a plugin is searched through the tags listed in its home page. This field contains one or more of these tags.
- *Version:* release of the plugin (e.g., 1.5.14).
- *Date:* release date (e.g., Sep 9 2014).
- *License:* the plugins license (e.g., MIT, GPL3, BSD…).
- *Maintainer:* name of the maintainer.

Table 9 includes additional information that could be useful for MORPHEMIC:

Table 9 All information provided by Jquery plugin registry.

Field	Description	Example
Block-tags	List of tags	* jquery lightbox <a class="tag icon-tag" *
block-version	List of versions realised for the plugin	*The list reports num.version, release-date, status of the plugin (it an additional information): 1.5.14 Version September 9, 2014 Released *
download	Reference download page	* <a class="download" href="http://github.com/jackmoore/colorbox/zipball/1.5.14">*
GitHub activity	Widget GitHub activity group, fork on GitHub provide the reference to the code of the plugin	*<aside class="widget GitHub-activity group"><h3 class="widget-title">GitHub Activity</h3> 3533 Watchers 931 Forks *
Author of the plugin	Who developed the plugin_name photo, avatar	*<aside class="widget author-info"> Author Jack Moore *
widget-licenses	License Information	*MIT*
widget-dependencies	Possible dependencies of the plugin	* Dependencies jquery >=1.3.2 *

4.3.4 Repositories associable to the Web Crawler

This section provides an analysis of other source repositories, currently not associated with the latest version of the Web Crawler, whose features can be potentially interesting for MORPHEMIC.
The analysis took into account several repositories. Some of them, namely:

- *Freecode*[28] is static (since 2014) and it is deprecated.
- *Java.net*[29] as reported in the official home page is closed (most of the projects have been reallocated).

The others, i.e., metaCPAN[30], OW2[31], CRAN[32] , CTAN[33] and r-forge[34] can be associated with MORPHEMIC by implementing the specific data fetcher to be included in the Web Crawler.
A specific case is CPAN[35], which although it has been online since 1995, the search operations for the stored data are performed through a new web interface called metaCPAN. For this reason, the analysis will be focused on metaCPAN.
The analysis provided in this section is similar to the previous sections for the associated repositories. Specifically, once the repository has been demonstrated as active and up to date, it will focus on:

[28] http://freshmeat.sourceforge.net/
[29] http://www.oracle.com/splash/java.net/index.html
[30] https://metacpan.org/
[31] https://www.ow2.org/
[32] https://cran.r-project.org/
[33] https://www.ctan.org/
[34] https://r-forge.r-project.org/
[35] https://www.cpan.org/

1. *the structure of the contained data and metadata* (xml, sql, json file or other),
2. *the information provided* which can be useful for application profiles.

Table 10 Source code repositories comparison

Name	Information Model	Status
Freecode	N/A	Inactive
Cpan	Json	Inactive
metaCPAN	Json	Active
CRan	Tar.gz packages (through R archive project: ftp://cran.r-project.org/incoming/archive/)	Active
Ctan	Hyper Text Language	Active
OW2	Hypertext Language	Active
r-forge	Hypertext Language	Active
Java-net	N/A	Inactive

The parameters used for the analysis are provided in Table 10. Following is the description of the table fields:

- *Name:* name of the repositories.
- *Information Model:* the standard format on which the information projects are provided (N/A is reported if the repositories are inactive).
- *Status:* if the repositories analysed is active or inactive.

In the next sections, we provide the analysis of the aforementioned active source code repositories candidates.

MetaCPAN

MetaCPAN is a web interface *for searching Perl modules, packages and applications on CPAN* (acronym of Comprehensive Perl Archive Network) which is based on the Comprehensive TEX Archive Network model). MetaCPAN is based on Elastic search. It provides a RESTful interface as well as the option to create complex queries. The searching of the package can be done by utilising the following fields:

- *Author*[36]: the list of the developers. The information is provided in a Json format file, as reported in the following example:


```
{
        "timed_out" : false,
        "took": 3,
        "_shards": { "total": 3, "failed": 0, "successful": 3},     "hits": { "hits": [{"_index": "cpan_v1_01",
                HUCKFINN","email":"huckfinn@cpan.org,"website": [],"asciiname" : ""},
        "_id": "HUCKFINN" },
        ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
        "total" : 14039,
        "max_score" : 1.0
        }
    }
```


[36] https://fastapi.metacpan.org/v1/author/_search

- *Distribution*[37]: name of the language distribution (e.g., Moose Perl distribution) where the module can be installed. The information is provided in a Json file format as reported in the following example for the Moose Perl distribution:


```
{"bugs":
        {"rt":
                {"open": 47,
                 "source": "https://rt.cpan.org/Public/Dist/Display.html?Name=Moose",
                 "patched": 0,
                 "active": "71",
                 "new": 12,
                 ....................
                    }
          },
        "name": "Moose",
         "river": {
              "total": 4544,
              "immediate": 3040,
              "bucket": 4 },
          "external_package" : {
             "Fedora" : "Perl-Moose",
             "Debian" : "Libmoose-Perl"}
        }
```


- *File*[38]: It provides a set of information about the CPAN such as download URL, rating, distribution, maturity (released, suspended,), version of the file, directory of the file, type of file (if deprecated, active) etc. The information is provided in a Json file format as reported in the following example:


```
{"cpan_v1_01": {
        "mappings":
                {"file":
                        {"dynamic": "false","properties":
                                {
                                 "version_numified" : {"type": "float"},
                                 "id": {"type": "string","index": "not_analyzed","ignore_above" :2048},
                                "status" : { "ignore_above" : 2048,"index" : "not_analyzed",
                                "type" : "string" },
                                "directory": {type":"boolean" },
                                "download_url" : {"type": "string","ignore_above":2048,
                                "index":"not_analyzed" },
                                        "date": {"type": "date","format"
                                    :"strict_date_optional_time||epoch_millis"},
                                 "module": {"type" : "nested", "include_in_root" : true,"dynamic" : "false",
                                "properties" : {.....
                                },.....
                        "path": { "type":"string", "index": "not_analyzed",.....},
                        "stat": {"properties": {"size": {"type": "integer"},"gid":{"type" : "long"}, ....
                        }
    } ......}
```


- *Rating*[39]: rating of the release. It provides also other information such as score, total score, distribution of the package, name of release etc. The information is provided in a Json file format as reported in the following example:

[37] https://fastapi.metacpan.org/v1/distribution/_search
[38] https://fastapi.metacpan.org/v1/file/_search
[39] https://fastapi.metacpan.org/v1/rating/_search

```
{"cpan_v1_01":
        {"mappings":
                {"rating":{"dynamic" : "false",
                        "properties":{"helpful" ..........},
                    "user":  .........}},
                    "dynamic": "false"},
                "release": { "ignore_above" : 2048,"index":"not_analyzed",
                    .....
                    },
                "details": {"properties":
                    {"documentation": {
                        "index": "not_analyzed",
                        "type": "string",
                        "ignore_above"}}
        ..................................
}
```

- *Release*[40]: name of the release. It also provides other information such as checksum, status, download URL, name of the release and so on. The information is provided in a Json file format: as reported in the following example for the Moose Perl extension:

```
{"deprecated": false,
 "date": "2020-07-21T19:04:06",
   "status": "latest",
   "name": "Moose-2.2013",
   "provides": [
        "Class::MOP::Method",    "Class::MOP::Method::Accessor",    "Class::MOP::Method::Constructor",
        "Class::MOP::Method::Generated",    "Class::MOP::Method::Inlined",
        "Class::MOP::Method::Meta",    "Class::MOP::Method::Wrapped",    "Class::MOP::Module",
        "Class::MOP::Object",    "Class::MOP::Overload",    "Class::MOP::Package",    "Moose",
        "Moose::Cookbook",    "Moose::Cookbook::Basics::BankAccount_MethodModifiersAndSubclassing",
        "Moose::Cookbook::Basics::BinaryTree_AttributeFeatures",
        "Moose::Cookbook::Basics::BinaryTree_BuilderAndLazyBuild",
        "Moose::Cookbook:Basics::Company_Subtypes",...]
.....
"download_url" : "https://cpan.metacpan.org/authors/id/E/ET/ETHER/Moose-2.2013.tar.gz",
"checksum_sha256": "df74dc78088921178edf72d827017d6c92737c986659f2dadc533ae24675e77c",
.....
"author": "ETHER",
 "dependency": [{"relationship": "recommends","module":"CPAN::Meta","version": "2.120900","phase":
        "test"}, {"version":"0.001","module": "Test::Fatal","relationship" : "requires", "phase" : "test"},
        {"phase":"test",.....
.....
ources": {"bugtracker" :
                    {"web": "https://rt.cpan.org/Dist/Display.html?Name=Moose", "mailto": "bug-
                Moose@rt.cpan.org"}, repository": { "url" : "git://github.com/moose/Moose.git","type" : "git",
                "web":"https://github.com/moose/Moose"}, "homepage":"http://moose.perl.org/"},
 "maturity": "released", "checksum_md5": "8267be7e7fbd9fc99730b78335d120a8",
"abstract": "A postmodern object system for Perl 5",
"tests": {"na" : 0,"unknown": 13,"fail": 1,"pass": 1599}
```

The Web Crawler should fetch the information provided by metaCPAN. To get the data, a dedicated data fetcher should be built.

[40] https://fastapi.metacpan.org/v1/release/_search

CTAN

The Comprehensive TeX Archive Network[41] contains all type of TeX material. Most of the software packages are open source, so they can be downloaded and used (e.g., MikTeX is, one of the most popular distributions of TeX). CTAN is currently active and maintained.
The CTAN allows access to the information database and retrieves it. Queries can be sent in the form of a RESTful service (HTTP GET or POST). The response in JSON contains:

- *list of Authors*[42]: each author is contained in a JSON object with a set of attributes:
  - *key* (mandatory), a unique id to identify the author;
  - *given name* (optional);
  - *family name* (optional) can be the organizational name;
  - *female* (Boolean, optional, default *false*) defines the gender of the author;
  - *died* (optional), indicates if the author is alive.

For security reasons, this object does not provide the email of the author. Here is an example of an author object:

```
{"key":"abrahams","givenname":"Paul W.","familyname":"Abrahams"}
```

- *list of Topics*[43]: each topic is represented by a JSON object with the following attributes
  - *key* (mandatory), a unique id;
  - *details* (mandatory) short description of the topics.

Here an example of the Topic objects:

```
{"key":"arabic","details":"documentation in and support for typesetting Arabic"}
```

- *list of Packages*[44]: each package is represented as a JSON object with the following attributes:
  - *key* (mandatory) the unique id;
  - *name* (mandatory) name of the packages (for example JSON/1.2/1.2);
  - *details* (optional) short description of the content of the package.

Here is an example of the package object:

```
{"key":"abc2mtex","name":"abc2mtex","caption":"Notate tunes stored in ABC notation"}
```

The Web Crawler should fetch all information from the list of authors, packages or topics through the use of the specific Restful API. A dedicated data fetcher should be built to get this information.

OW2

OW2[31] is an independent no-profit organization dedicated to open source software and infrastructure. OW2 provides a Marketplace on which approved projects are published. The approval of new projects and their whole lifecycles are supervised by the OW2 Technology Council. The Marketplace also provides some filters to select the projects according to maturity level, functionalities, standards and licenses.
The list of projects is provided by the OW2 Project Repositories[45]. Specifically, this page provides only the name and the reference link of the project. The wiki of the OW2 provides other information, in particular:

- *spaces*[46]: basic information on the project, i.e., name and home page;
- *classes*[47]: properties of the project, including branch information, in a standard template;

[41] https://www.ctan.org/
[42] https://ctan.org/help/json/1.2/authors
[43] https://ctan.org/help/json/1.2/topics
[44] https://ctan.org/help/json/1.2/packages
[45] https://projects.ow2.org/view/ow2/ProjectRepositories
[46] https://projects.ow2.org/rest/wikis/projects/spaces
[47] https://projects.ow2.org/rest/wikis/projects/classes

- *modifications*[48]: the history project summary, including its major and minor versions.

The information can be provided by the project directories list in XML as reported in the following example[49]:

```
<wikis xmlns="http://www.xwiki.org">
<link href="https://projects.ow2.org/rest/wikis/query" rel="http://www.xwiki.org/rel/query"/>
<wiki>
<link href="https://projects.ow2.org/rest/wikis/projects/spaces" rel="http://www.xwiki.org/rel/spaces"/>
<link href="https://projects.ow2.org/rest/wikis/projects/classes" rel="http://www.xwiki.org/rel/classes"/>
<link href="https://projects.ow2.org/rest/wikis/projects/modifications" rel="http://www.xwiki.org/rel/modifications"/>
<id>projects</id>
<name>projects</name>
```

The main information provided for each project are website, functionality, status, license(s), VCS repositories, issue tracker URL and OW2 submission. As an example, the table below provides the information on the project ADR[50]:

Table 11 How the metadata information is provided for OW2

Example of OW2 information project	
Web site	https://projects.ow2.org/view/adr/
Functionality	Application platform
Status	Incubation
License(s)	GNU General Public License v2.0 only
VCS repository(ies)	https://gitlab.ow2.org/stsisi/adr-app/adr
Issue tracker URL	https://gitlab.ow2.org/stsisi/adr-app/adr/issues
OW2 submission	ADR app

The Web Crawler should fetch the information provided by the OW2 Project Repositories. To get the data, a dedicated data fetcher should be built.

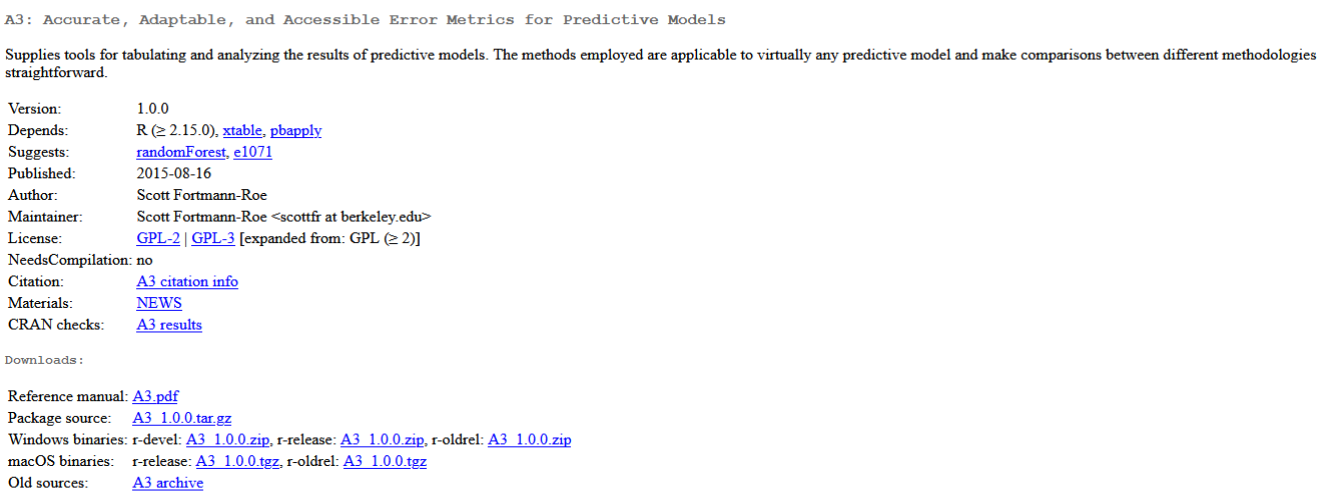
CRAN project

CRAN (Comprehensive R Archive Network) provides modules written in R. CRAN is a network of FTP servers and web servers that offer the updated version of R, along with documentation and additional modules.
The CRAN package repository contains 16433 packages; the packages are available and sorted by date of publication or by name[32].
The FTP site[51] enables to download the software. The information listed below is provided in HTML form or in plain text:

- *description* of the project;
- *name* of the project;
- *version* of the release project;
- *date* of publication;
- *author* of the packages;
- *maintainers* of the packages (one or more users);
- *URL link* to the project;
- *packages downloads*: it is a reference link to download of the packages of the projects.

Figure 18 Figure 18 Example of information provided by cRanshows an example how the information provided.

[48] https://projects.ow2.org/rest/wikis/projects/modifications
[49] https://projects.ow2.org/rest/wikis/projects/#list
[50] https://projects.ow2.org/view/adr/
[51] ftp://cran.r-project.org/pub/R/



A3: Accurate, Adaptable, and Accessible Error Metrics for Predictive Models

Supplies tools for tabulating and analyzing the results of predictive models. The methods employed are applicable to virtually any predictive model and make comparisons between different methodologies straightforward.

Version:	1.0.0
Depends:	R (≥ 2.15.0), xtable, pbapply
Suggests:	randomForest, e1071
Published:	2015-08-16
Author:	Scott Fortmann-Roe
Maintainer:	Scott Fortmann-Roe <scottfr at berkeley.edu>
License:	GPL-2 | GPL-3 [expanded from: GPL (≥ 2)]
NeedsCompilation:	no
Citation:	A3 citation info
Materials:	NEWS
CRAN checks:	A3 results

Downloads:

Reference manual:	A3.pdf
Package source:	A3_1.0.0.tar.gz
Windows binaries:	r-devel: A3_1.0.0.zip, r-release: A3_1.0.0.zip, r-oldrel: A3_1.0.0.zip
macOS binaries:	r-release: A3_1.0.0.tgz, r-oldrel: A3_1.0.0.tgz
Old sources:	A3 archive

Figure 18 Example of information provided by cRan

In order to use CRAN as source repository for MORPHEMIC, it should be evaluated how to extract and how to use the information provided.
The data fetcher should process the DOM HTML used to provide these metadata.

R-forge

As reported in the R-forge homepage[34], RForge provides a collaborative environment for R's developers. It provides a SourceForge-like services (such as SVN repository, place for documentation, downloads, mailing lists, bugzilla, wiki etc.) with additional features specific for the development of R packages, such as check on-commit, nightly builds of packages, testing on various platforms and full CRAN-like repository access. It is complementary to sites like GitHub with which it can integrate as R package back-end.
R-Forge is based on FusionForge through which it has easy access to the SVN repository, packages compiled and checked daily, mailing list, bug tracking, and so on. The projects are grouped by category[52] (by default the project is grouped in the Topics category) as:

- *Topics:* e.g., Bayesian statistics, bioinformatics.
- *Development Status:* e.g., for beta, alpha release.
- *Environment:* e.g., win32, console, another environment.
- *Intended Audience:* e.g., developers, end users.
- *Natural Language:* e.g., English, French, Korean.
- *Operating System:* e.g., BeOS, MacOS, Microsoft.
- *License:* e.g., Public Domain.
- *Programming Language:* e.g., C/C++, Java, other.

The standard used to provide all the information contained in an R-Forge project is HTML format. The information provided is:

- *name* of the project;
- *description* of the project;
- *project information* provided for each category (Environment, Topics, Intended Audience, and so on);
- *project member:* list of the members of the project (developers, projects manager, testers, and so on);
- *project tools:* reference link to the project home page.

The following figure shows an example on how the information is grouped, collected and provided for PTauxPC[53]:

[52] https://r-forge.r-project.org/softwaremap/trove_list.php
[53] https://r-forge.r-project.org/projects/ptauxpc/

Figure 19 Example how the information is provided

The Web Crawler should fetch the information provided by R-forge by processing the DOM HTML model (used to provide these metadata).

4.3.5 Considerations on the available resources to the MORPHEMIC's Web Crawler

The MORPHEMIC's Web Crawler retrieves metadata from pre-defined repositories of open source software, such as forge, metaforge, and list of directories.
The advantage of using pre-defined repositories is to restrict the search scope to a specific region of the web and specific topics, preserving both computational and communication resources (such as network resources, server overload, server and router crashes, network and server disruption).
In particular, concerning the three repositories currently used (GitHub, Apache, jQuery Plugin Registry) the possibility to retrieve the same project on different repositories increases dramatically the quality and the reliability of the retrieved information. Specifically, it is possible to find different pieces of information on the same fields, but also different fields on different repositories. In both cases the integrated version provides better quality. The second case, in which different fields come from different repositories, is very important to provide the Code Analyser with all the needed information. The following table analyses this aspect by indicating which of the three considered repositories provides information for each field common and uncommon. In addition, the tables provide the common fields that are not always mapped for specific project (common fields as release, repository-location, mailing-list).

Table 12 Metadata obtained by the preliminary analysis performed by the WebCrawler

Name	Description	GitHub	Apache	jQuery Plugin
Name	The name of the project.	Yes	Yes	Yes
Shortdesc	The short (8 or 9 words) plain text description of a project.	It could be	It could be	No
Description	The plain text description of a project, of 2-4 sentences in length.	Yes	Yes	Yes
Homepage	The link to the homepage of the project.	Yes	Yes	Yes
Created	The date of the creation of the project.	Yes	Yes	Yes
Source	The source of Information.	Yes	Yes	Yes
Revision	The revision identifier of a software release.	Yes	Yes	Yes
Old homepage	If the DNS changed.	No	Yes	No
Service endpoint	The URI of a web service endpoint where software as a service may be accessed.	Yes	Yes	Yes
Release	The release of the project.	Depending on the event type	Yes	Yes
Repository-location	The repository link where the source code can be downloaded.	Depending on the event type	Yes	Yes
Bug database	The bug tracker for a project.	It can be retrieved	It could be	No
Category	The category of the project.	No GHArchive	It could be	It could be
Download page	The web page from which the project software can be downloaded.	It can be retrieved	It could be	It could be
Download mirror	The Mirror of software downloads web page.	Depending on the project	Yes	Yes
Wiki	The URL of Wiki for collaborative discussion of project.	Depending on the project	Yes	Yes
Programming language	The programming language a project is implemented in or intended for use with.	No	It could be	No
OS	The operating system that a project is limited to. Omit this property if the project is not OS-specific.	No GHArchive	It could be	No
Language	The ISO language code project has been translated into.	No GHArchive	Yes	No
License	The URI of an RDF description of the license the software is distributed under.	No GHArchive	It could be	It could be
Developer name	The developer of software for the project.	Yes	Yes	Yes
Mailing_list	The mailing list home page or email address.	It depends on the event type	No	It could be
Platform	The indicator of software platform (non-OS specific), e.g. Java, Firefox, ECMA CLR.	No GHArchive	Yes	Yes
Audience	The description of a target user base.	No GHArchive	No	No
Blog	The URI of a blog related to a project.	No GHArchive	No	No

Taking into account only the three aforementioned repositories, the common fields are:

- *Name* of the project.
- *URI* home page of the project.
- *Description* of the project.
- *License.*
- *Developer name.*
- *Date of creation* of the project.

A set of fields are common to Apache and jQuery Plugin Registry. For GitHub (Section 4.3.1) they can be provided depending on the specific event associated, in particular PushEvent or PullRequestEvent. The fields are:

- *Release* version.
- *Supported Languages.*
- *URL of the repositories* where the source code can be downloaded.
- *Service endpoint.*

The integration of these three repositories can potentially provide the minimum information set required by the Code Analyser.
A further selection could be made by comparing the crawled projects with the MORPHEMIC's use cases and rely on the analysis techniques that we based on. This comparison should allow us to understand which and if there are attributes to add to those already selected by the MARKOS crawler.
Another important aspect, towards selecting additional repositories to integrate, concerns the format of the information, which impacts on the data fetcher. All the analysed repositories (yet implemented in Web Crawler) provide information in different formats. Simple and structured formats, such as XML and JSON simplify the work of the data fetcher, while HTML requires more complex implementations. The HTML format is used in some of the repositories not associated yet. This is another reason for carefully evaluating costs and benefits to develop data fetchers and to identify the fields needed for the Code Analyser.
This analysis started from what is provided by all the repositories taken into account, both the associated ones and not-associated ones. The common fields are the following:

- *Name* of the project.
- *URI* home page of the project.
- *Description* of the project.
- *Date of creation* of the project, plugin module.
- *Developer name.*
- *License.*

The Code Analyser is still in the phase of design (Section 3.4) and the information needed to enhance the application profile needed by MORPHEMIC should still be finalized. This impacts on the selection of the repositories that cannot be finalized until the respective information needed is supplied.
However, some preliminary considerations can be conducted, for example the license seems not so useful for providing a deployment model (although it could be useful to select only projects which have a specific license, for example open-source license) , while the date of release could indirectly help (a selection criterion).
In this case, the analysis starts with what is provided by the repositories. It is also important to start from the application profile and define if any information not common to all the repositories or not provided at all may be useful. A help in collecting this information could come from the Code Analyser. For example, supported languages might be derived this way, although not provided by a repository. Another difficulty is that while a repository has the information model to provide a field, the value for that field may be missing for a specific project.
In these cases, it is critical to find a way to obtain this information (directly or indirectly) from the data provided by the associated repositories (and the respective code associated with them). This analysis is not easy, and the successful result is not guaranteed, but it is very important to perform it very carefully to get as much information as possible and to define the functionalities that MORPHEMIC will provide.

5. Review of Code Analysis & Classification

Code mining will allow us to identify a corpus of software project code that is representative for *code classes* like High Performance Computing (HPC) code or web code. The different classes are characterised by certain *features* that can range from the qualitative features like the programming languages used, structural features reflected in the application call graph, i.e., the way components and functions invoke other components and functions, and the data structures in use (e.g., fixed sized arrays or dynamically scaling vectors or lists). Furthermore, all computer software is about processing data, and so the data processing graph where some components process the data before others will also reveal the type of application. In order to successfully classify an application code base given to MORPHEMIC, the following steps are essential:

- *Feature definition*: This step is necessary in order to identify common traits of various code types. The features to be collected must be sufficient to differentiate one code class from another code class, and they must be generic and available in all code classes. Some features can be ordinal, like the programming language or the used data structures; some features are quantitative like the number of code lines or the number of functions, classes, or components; some features can be structural and mixed, like the software patterns identified in the code, and how many times each pattern is encountered. The code corpus is not available at the time of writing and so it has not been possible to identify the concrete features to use in the MORPHEMIC code mining at this stage; thus, defining good features is a major challenge for the next period.
- *Feature extraction (measurements)*: After identifying the features to use to differentiate the code classes, it is necessary to be able to measure the features automatically. There is no point in defining a feature which cannot be automatically measured. Hence, the initial focus of the work on classification documented in this report has been to identify tools and mechanisms that can be used to gather information about the code of each software project in the code base. The result of the feature extraction is a vector of values in the individual feature dimensions, $\vec{x}$ used to characterize the analysed code, and to differentiate among different code classes. Section 5.1 discusses the various approaches and interesting tools that can be used for feature extraction, to be tested and applied on the code corpus in the next phase.
- *Classification*: The final step is then to look at similarities of the features of the application code to be deployed by MORPHEMIC with the application code classes identified in the code corpus. Identifying the code classes in the code corpus can either be done manually and *a priori*, or automatically by grouping together code whose feature vectors are similar according to some distance measure. The code to be deployed can then be said to belong to the class to which its feature vector is most similar according to the same measure. The exact meaning of distance and similarity in classification is further discussed in Section 5.2 providing the baseline to be applied when the automatic feature extraction has been established.

5.1 Techniques for static code analysis

5.1.1 Static code analysers

Static code analysis [5] normally deals with detecting code issues or vulnerabilities, but also code flow visualisation and dependency detection. In static analysis, the code under examination in not executed.
There are many tools available[54] and it may be worth considering the adoption of some existing static code analysers. The most promising open source tools are listed below:

- *Coccinelle*: open source, pattern matching and transformation tool that works only for C/C++[55]. It can be used for pattern matching.
- *ConQAT*: open source, software quality analysis engine developed by Technical University in Munich and CQSE Company. It provides visualisations, similarity detection, and it supports many software languages. It has not been supported since 2018, because the company commercialized it as a new product named Team scale. However, it can be considered as a good starting point for future research.[56]

[54] https://en.wikipedia.org/w/index.php?title=List_of_tools_for_static_code_analysis&oldid=987801100
[55] https://coccinelle.gitlabpages.inria.fr/website/download.html
[56] https://www.cqse.eu/en/news/blog/conqat-end-of-life

- *Frama-C*: open source program analyser for C. It enables the slicing of a program into smaller parts, detects spare code and computes dominators of statements[57].
- *Moose:* Free and open source platform for software and data analysis. It provides meta-modelling and it is designed to work on any data. It provides graph visualisations, browsers to search the source code, parsers, models[58].
- *PrettyDiff:* open source data comparison tool which can compare two pieces of the source code[59].
- SonarQube: open source platform for inspection of code quality, but also reports on duplicated code, code complexity and comments[60].
- *SourceMeter:* open source code analyser tool which performs deep static program analysis. It constructs abstract semantic graphs and it calculates product metrics[61].
- *Squale:* open source platform for multi-language applications. It provides basic monitoring code data such as number of lines, classes or the level of maintainability of the code[62].
- *Yasca:* open source program which reports code-quality related metrics[63].

5.1.2 Code quality checkers

Code quality analysis and audits have become an essential process for engineering software systems. In particular, with the increasing use of open source software, security and other code quality parameters have become critical in developing high quality software. Software quality can be assessed based on two related aspects:

- Software *functional quality* refers to how well it complies with or conforms to a given design based on the predefined functional requirements or specifications. It can also be described as a parameter to measure the degree to which the correct software was produced, as well as to compare a piece of software to competitors in the marketplace.
- Software *structural quality* refers to how it meets non-functional requirements, such as reliability, robustness or maintainability. This type of quality is more diverse with respect to the type of software, users, and the deployment conditions.

The above aspects are rather high level, identifying the main categories for code quality analysis. From a more detailed perspective, the code quality can be assessed based on the following indicators:

- *Readability*: readable, no useless code, brevity/conciseness, formatting/layout, style, indentation, naming convention.
- *Structure*: well-structured, modular, cohesion, low coupling, no duplication, decomposition.
- *Testability*: testable, test coverage, automated tests.
- *Dynamic behaviours*: robust, good performance, secure.
- *Comprehensibility*: understandable, clear purpose.
- *Correctness*: runnable/free of bugs, language choice, functionally correct (meeting business requirements).
- *Documentation*: documented, commented.
- *Maintainability*: maintainable, adaptable, reusable, used by others, interoperable, portable.
- *Miscellaneous*: license, suitable data structure, metrics/measurements.

There have been many projects trying to assess the code quality. In the following, we list the most popular tools for code quality checking:

1. *SonarQube (open-source):* As mentioned above (Section 5.1.1), it can inspect code quality. SonarQube is one of the most popular code quality and security analysis tools. It can check many Correctness aspects of code, including variable declarations, exception handling, and detecting potential bugs and complex code. It

[57] http://frama-c.com/
[58] http://moosetechnology.org/
[59] https://github.com/prettydiff/prettydiff/
[60] https://www.sonarqube.org/
[61] https://www.sourcemeter.com/
[62] http://www.squale.org/
[63] https://www.scovetta.com/yasca/

supports over 25 programming languages, which is a higher language support level than most tools in the market.

2. *Kritika (closed source):* is an online code analysis tool that analyses public and private repositories. It can analyse the code for coding standard violations, security threats, test coverage, and complexity of the code (Readability, Structure and Correctness indicators). It is integrable with GitHub to display code quality statistics. It supports more than 12 programming languages and text files.[64]
3. *CodeSonar (open source):* it is a code analysis tool that analyses the code from a computational perspective. It is able to develop models from your code, which can analyse mainly Correctness, namely potential execution threats like deadlocks, memory overflow, null pointers, data leaks, and other programmatic errors that might be difficult to discover. It supports C, C++, C#, Java, Python, and binary code of Intel x86, x64 and ARM[65].
4. *JArchitect (closed source):* it is dedicated to code analysis in Java. It is one of the most exhaustive Java code analysis tools that analyses mainly Structure and Correctness aspects, namely call hierarchies, memory consumption, code complexity, functional coupling, block nesting depth, and architectural flaws in the code[66].
5. *Code Climate (open source):* it is an analytics tool that offers two different products: 1) *Velocity:* it focuses on improving the functional quality of the code, and in particular on the Structure quality indicator. It identifies logical flaws and bad design patterns within the code and then provides a visualization of the code quality analysis and guidelines for solving the discovered issues; 2) *Quality:* focuses mainly on Readability and Correctness quality indicators, including formatting, unused imports, variables, and unit test coverage. Code Climate supports more than ten languages[67].
6. *Fortify by Micro Focus (closed source):* it focuses on analysing security vulnerabilities in the code which are related to Correctness. It scans known security flaws and any presence of malware or corrupt files. Some of its features include automated scanning of code supporting almost every programming language, and providing suggestions for fixing vulnerabilities as the result of analysis[68].
7. *Codecov (open source):* it analyses mainly code Correctness quality and bugs, scans in for security checkers, and monitors the popular trends across the developer community. The languages supported include: Java, JS, Node, Python, Go, Ruby, Swift, Dart, Haskell, and others[69].
8. *Codacy (closed source):* it allows automated checking of potential security risks in the code, styles guide misinterpretations, analyses the code against best code practices, and even supports code coverage to see how much your tests are covering[70]. As indicated, this checks mainly the Correctness and Testability quality aspects.
9. *Zoompf (closed source):* it is an enterprise-level performance audit platform for integration within the app and mobile app development workflows. It audits the code to understand the root issues of slow performance and what can be done to fix them.[71] Therefore, Zoompf is focused on Dynamic Behaviours quality indicator.

5.1.3 Graph visualization for matching

Software visualization refers to the visualization of artifacts related to software and its development process.
Generally, three different aspects of a software system can be visualized:

- *Structure*: refers to the static parts of the code and relations. The structural visualization includes the program code and data structures, the static call graph, and the organization of the program with respect to its constituting modules.
- *Behaviour*: refers to the execution of the program with real and abstract data. The execution can be described as a sequence of program states. A program state contains both the current code and the data of the program. Depending on the programming model and the target language, the execution can be visualized at a high level of abstraction as functions calling other functions, or as objects communicating with other objects.
- *Evolution*: refers to the adaptation and reconfiguration in a software system and, in particular, emphasizes the fact that program code may need to change over time to extend the functionality of the system or to remove software bugs and failures. Software evolution can be visualized from three different aspects: visualizing

[64] https://kritika.io/
[65] https://www.grammatech.com/products/codesonar
[66] https://www.jarchitect.com/
[67] https://codeclimate.com/
[68] https://www.microfocus.com/
[69] https://codecov.io/
[70] https://www.codacy.com/
[71] https://zoompf.com/

changes in software metrics, visualizing software archives and histories, and visualizing software structural changes.

As mentioned above, in MORPHEMIC we mainly focus on static code analysis. The reason for considering code graph visualization in this section under the subject of static code analysis is that the code graph can be matched against prototype application patterns (software or architectural or deployment patterns).

Hence, for code visualization, we consider structural visualization which can be textual or graph-based as explained below:

- *Textual Representation:* this refers to how to present the program code, which includes printable and non-printable text (e.g., blank and line feed). The common practice with textual representation is pretty printing. The goal of pretty printing is to make the nesting of code blocks visible while using a minimal number of lines for each block.
- *Diagrammatic Visualization:* diagrams have been used to show the structure of code. In these diagrams relations between program parts are visually encoded by actions (to represent a code block), edges (to indicate which function invokes which other function), neighbourhood (i.e., alternative actions that are often placed next to each other), and containment (a box representing a complex action contains the boxes of its sub-actions). There exist four general diagrammatic-based representations of code, including:
    - Jackson Diagrams: In this model, the data structures involved are first hierarchically decomposed, and then the program structure should follow this decomposition. The basic elements of Jackson diagrams are actions, which can be decomposed into sub-actions, as shown in the following Figure:

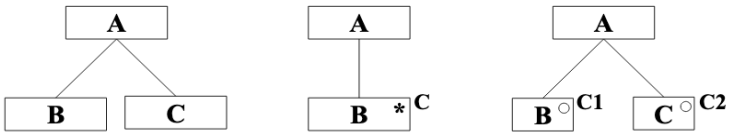


Figure 20 Jackson Diagram

A sequence A consists of the execution of a sub-action C after a sub-action B. An iteration A consists of multiple repetitions of B as long as an iteration condition C holds. Finally, an alternative A is either a sub-action B if a condition C1 holds or a sub-action C if a condition C2 is true.

- Control-Flow Graphs: In these graphs, rectangular represent events, activities, processes, functions, or statements, whereas diamond nodes show branch conditions and can have several exits. Edges in the graph depict transitions from one statement to another, i.e., the flow of control as shown in the following Figure:

Figure 21 Control Flow Graphs

- Nassi–Shneiderman Diagrams: introduced nested rectangular diagrams, also known as *structograms*. The primitive diagrams are shown below:

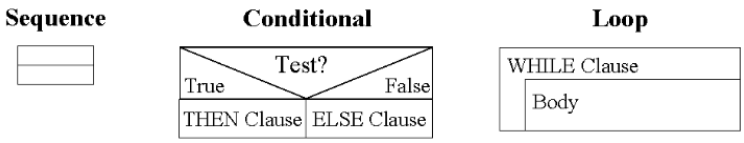


Figure 22 Nassi-Shneiderman Diagrams

- Control-Structure Diagrams: are for keeping the sequential order of the program parts in the source code. They make the nesting and scope of program constructs more explicit through a horizontal tree. Vertical lines show the extent of blocks, and vertically stretched oval lines show that of loops. Diamonds represent conditional statements (as reported in the Figure 23).

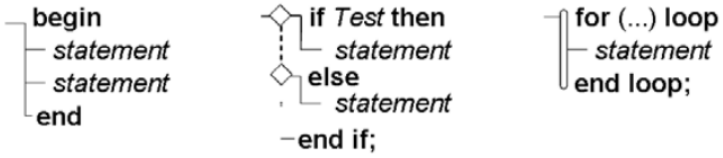


Figure 23 Control-Structure Diagrams

- *Visualizing Software Architectures:* Visualizations of software architectures mainly illustrate the code structure at various levels of abstraction. At a high level, the architecture consists of components with ports/interfaces, and ports are linked through connectors. Besides this, there are many other architecture-related aspects, such as the global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives. Most of these aspects have both functional and non-functional properties.
- *The Unified Modeling Language* (UML): UML offers a number of different types of diagrams, including use case diagrams, class and object diagrams, behaviour diagrams (state chart diagrams and activity diagrams), interaction diagrams (sequence diagrams and collaboration diagrams), implementation diagrams (component diagrams and deployment diagrams), and model-management diagrams (packages, subsystems, and models). A class diagram is perhaps the most common type of diagram in UML which represents the classes, their interfaces, properties, and their communication with other classes in the software system.

5.1.4 Code analysis techniques

Static analysis techniques are the most popular choice for analysis of software code as they are very simple and fast. In the following, we discuss some well-known techniques for static code analysis [6] [7].

- *Syntactic Pattern Matching.* This technique is based on the syntactic analysis of the code by a parser. The parser takes the source code as input and generates a data structure called abstract syntax tree. One usage of this technique is bug finding. Using this technique, a set of program constructs that are potentially dangerous or invalid are defined, and then the target program's abstract syntax tree is searched for instances of these pre-defined constructs. Syntactic pattern matching is considered the fastest and easiest technique for static analysis. However, it may provide little confidence in program correctness resulting in many false alarms.
- *Data Flow Analysis.* This is a popular static analysis technique in which a graph-based representation of the code is extracted, called control flow graph, and then dataflow equations for each node of the graph are written. Then, the equations are repeatedly solved to calculate output from input for each node locally until the equations stabilize or reach a fixed point. The main dataflow analyses include reaching definitions (i.e., most recent assignment to a variable), live variable analysis (i.e., removing unused assignments), and expression analysis (i.e., elimination of redundant arithmetic expressions).
- *Abstract Interpretation.* It is based on a theory of semantics approximation of a source code based on monotonic functions over ordered sets. Given a programming language, abstract interpretation consists of giving several semantics linked by relations of abstraction. A semantics denotes a mathematical description of the behaviour of the program. The most precise semantics, describing very accurately the actual execution of the code, are called the concrete semantics. For instance, the concrete semantics of an imperative programming language may associate to each program the set of execution traces it may produce. Then, more abstract semantics can be derived, e.g., one may consider only the set of reachable states in the executions. The goal of this analysis technique is to derive a computable semantic interpretation at some point.
- *Constraint-Based Analysis.* A constraint-based analysis traverses the code to emit and solve constraints describing properties of a program. This technique is broken into two steps. First, it produces constraints from the program text, which describe the information or behaviour desired from the program, called constraint generation. The second step is dedicated to solving the constraints by computing the desired information, called constraint resolution. Static information is then extracted from these solutions. One key feature of this technique is that algorithms used for constraint resolution can be written independently of the target constraint system.

5.1.5 Software and Patterns

In software engineering, a design pattern is a general and reusable design solution to the design problem which may occur frequently during the software design and implementation phases. It should be noted that a design pattern is basically proposed at a relatively abstract level, meaning that a design pattern is not, e.g., an algorithm, which can be converted directly into code. Rather, a design pattern is a *template* for solving a general problem. Design patterns provide a number of advantages to the software development process, including reusability of software and design, documentation (allowing developers to recognize the structure and design of the software), as well as communication and teaching (providing a common language for software designers and developers and improving the communication between them). For example, in object-oriented programming, design patterns can increase the reusability of the software libraries and accelerate the development process with proven successful development patterns.

Mining design pattern instances from the source code can significantly help to understand the code and its structure and change it over the software lifetime. It can also help in facilitating the discovery of code similarity between source code. Through an accurate and efficient mining solution, we can extract the used design patterns in the code, and this will be the basis for inferring similar source codes. However, in typical software programs, several patterns may be combined or offered as alternatives. For example, the composite, iterator, and visitor constructs are often used in combination, while the prototype pattern may be used as an alternative to the abstract factory. This, therefore, calls for mining a sequence of patterns when code similarity is the purpose of mining. It should be noted that another application of pattern sequence mining is the extraction of strong and weak relationships between the design patterns used in the code (e.g., in object-oriented source code), which will enable analysers and programmers to determine the dependency rate of each object, software component, and other parts of the code for parameter passing and modular programming.

Sequence alignment is a popular method of discovering the similarity between two sets of data. It can be divided into two sub-methods: double alignment and multiple alignments. Sequence alignment has been widely used in bioinformatics for genome sequence analysis and difference identification. Any sequence of DNA (Deoxyribonucleic acid), ribonucleic acid (RNA), or proteins can be aligned using various bioinformatics algorithms. Sequence mining is basically one type of data mining to statistically identify the pattern in a set of input data. The pattern values are generally assumed to be discrete. DNA sequence mining is a method for finding the common subsequence in a set of sequences.

Each design pattern has specific properties and characteristics while it might propose to use classes or components with specific variable names and parameters, but programmers may change such names. A pattern can be converted to a metric form so that the structural design pattern design of a given source code can be extracted based on the code variables, parameters, and methods [8]. After the conversion to metric code, the source code is searched for each programming pattern using the DNA sequence alignment method, which is implemented using dynamic programming [9]. Then, the DNA sequence method is used to identify the largest match between each pattern and a specified section of the source code. Each design pattern is compared with each part of the source code, resulting in sequence alignments of various degrees as output. Then, all the sequence alignments are analysed to find the best match between the design pattern codes and source codes using DNA sequence alignment. If a given section of the source code overlaps significantly with a specified design pattern compared to other patterns, then that source code section is labelled with the matching design pattern.

Using the above approach for finding code similarity, sequences of software patterns can be the code DNA and software can be compared based on the similarities of these sequences.

5.1.6 String matching

Text-based approaches [10]–[11] apply string matching techniques, e.g., Longest Common Subsequence (LCS) over two string sequences of code. They are more efficient when comparing identical code while their accuracy drops with the existence of syntactical and semantic changes on the compared code. Some, however, were able to bypass the syntactic differentiation problem, especially its variable renaming instance [11]. Apart from LCS, other string-matching techniques have been employed as in the case of PMD [12]. Token-based approaches [13]–[14] transform a string sequence of a code into a set of words to represent a certain program. By adjusting the type of tokens to be employed, the programs can be abstracted in such a way that textual differences can be normalized. This line of work is able to tolerate added or deleted statements and bypass formatting and lexical differences but has higher time complexity than the others. However, approximations or optimizations [15] can be used to reduce this complexity. We opt out here metrics-based approaches, as these lead to low accuracy results. The result of the feature extraction is a vector of values in the individual feature dimensions, used to characterize the analysed code, and to differentiate among different code classes.

5.2 Algorithms for classifying the code

Code similarity techniques can be categorized [16] into metrics-based, text-based, token-based, tree-based, graph-based, and pattern-based. Metrics-based approaches [17] rely on metrics or software measures (e.g., Halstead complexity measures) but have been found not to be so effective [18] in terms of the other approach categories.
Optimally, the features should be defined such that the feature vector for each class is a *standard basis unit vector* with full weight on one feature, and zero weight on all the others features. However, real code will never score only in one feature dimension. Consider, for instance, code processing big data sets in parallel: This can have strong High-Performance Computing (HPC) elements in the way it does parallel processing; big data aspects in the data handling; and show similarities with multimedia code in its stream processing of data. It is therefore necessary to have a *distance* metric to decide if the code should be treated as HPC, big data, or multimedia code.

There are many different distance metrics proposed for various purposes, and the choice of a distance metric both depends on the type of data, and how the code classes are represented [19]. Consider, for instance, the situation where the code is characterized by a set of *words* to represent its class, i.e., think of it as a 'world cloud' that could be manually annotated as a result of an evaluation conducted by software developers. Such *ordinal* data can be compared using the *Jacquard distance* measure that assesses the dissimilarity between two sets $\mathbb{A}$ and $\mathbb{B}$

$$D_J(\mathbb{A},\mathbb{B}) = \frac{|\mathbb{A}\,\Delta\,\mathbb{B}|}{|\mathbb{A}\cup\mathbb{B}|} = \frac{|\mathbb{A}\cup\mathbb{B}| - |\mathbb{A}\cap\mathbb{B}|}{|\mathbb{A}\cup\mathbb{B}|}$$

In the *cardinal* case the feature vector $\boldsymbol{x}$ provides information about the score of the code in each direction where it can be assumed, without any lack of generality, that each feature score is a real number over the unit interval, i.e. $x_i \in [0,1]$ measuring the strength by which the code has the particular feature of dimension *i*. Classification must then be done by comparing the feature vector $\boldsymbol{x}$ with a characteristic feature vector for a code class *C* represented by $\boldsymbol{x}_C$. The distance metric used for the classification must then be able to clearly distinguish between the different classes; in this case, the code having the feature vector $\boldsymbol{x}$ belongs to the code class C for which it has the least distance to $\boldsymbol{x}_C$.

There are situations where it is possible to decide *a priori* the characteristic feature vector $\boldsymbol{x}_C$ of code belonging to the code class C. The natural choice for a distance metric is then the *Euclidean norm*,

$$D_E(\boldsymbol{x}|\boldsymbol{x}_C) = \|\boldsymbol{x}-\boldsymbol{x}_C\|_2 = \sqrt{(\boldsymbol{x}-\boldsymbol{x}_C)^T(\boldsymbol{x}-\boldsymbol{x}_C)} = \sqrt{(x_1-x_{C,1})^2+(x_2-x_{C,2})^2+\cdots+(x_n-x_{C,n})^2}$$

Alternatively, one may use to *Soergel distance* using normalized absolute differences in each feature dimension instead of the squared distances of the Euclidean norm [19],

$$D_S(\boldsymbol{x}|\boldsymbol{x}_c) = 1 - \frac{\sum_{i=1}^{n}\min(x_i,x_{C,i})}{\sum_{i=1}^{n}\max(x_i,x_{C,i})} = \frac{\sum_{i=1}^{n}|x_i - x_{C,i}|}{\sum_{i=1}^{n}\max(x_i,x_{C,i})}$$

In most cases, one cannot define theoretically the characteristic feature vector of a code class, and it is necessary to calculate it statistically from a set of code samples for which the class is already known. The natural choice for the typical feature vector of this class is then the arithmetic average feature vector for the known elements belonging to this class, $\bar{\boldsymbol{x}}_C$. The Euclidean norm then generalizes to the *Mahalanobis distance* [19],

$$D_M(\boldsymbol{x}|\bar{\boldsymbol{x}}_C,\boldsymbol{S}_C) = \sqrt{(\boldsymbol{x}-\bar{\boldsymbol{x}}_C)^T\boldsymbol{S}_C^{-1}(\boldsymbol{x}-\bar{\boldsymbol{x}}_C)}$$

where $\boldsymbol{S}_C$ is the sample covariance matrix of the feature vectors for the code samples that are known to be elements of the class C. Again, the code with feature vector $\boldsymbol{x}$ is taken to belong to the class C for which its Mahalanobis distance is minimum.
Classification based on distance measures is normally strong in the situations where it can be used, although it should be observed that finding the optimal classification based on distance minimisation is a combinatorial optimisation problem that is NP-hard [20]. The Mahalanobis distance requires a training set of various code types manually classified and labelled. Once a new code has been successfully classified, it can be added to the training set and the

average vector and the covariance matrix of that class can then be updated to include the new code sample. In this case, the classification will grow more robust to random variation in the feature vectors of the classes over time. However, it is difficult to add a new class of code when the characteristic vectors are calculated from a training sample. In this case, it could be that one of the samples already classified as belonging to one of the old classes will have a shorter distance to the new class, and one would need to return to the original training set to re-calculate the code class averages and covariances, followed by a subsequent re-classification of all code samples previously classified.

5.2.1 Tree and Graph based methods

Tree-based code similarity tools [21] rely on transforming code into internal, normalized representations like abstract syntax trees which are then compared to find similar or common subtrees. Then one can apply different similarity measures like suffix trees [22], for which optimal algorithms exist [23], or the Jaccard similarity coefficient [24], i.e. $1 - D_J(\mathbb{A},\mathbb{B})$, over these latter sets to infer the similarity between two software programs. Graph-based approaches [25], [26] cover both the structure and the semantics of the code but also suffer from the problem of increased time complexity. In fact, most graph matching algorithms are NP-Complete. As such, they also suffer from scalability problems. Specific types of graphs are usually exploited, such as Program Dependence Graphs (PDGs) [26] and Control Flow Graphs (CFGs) [25], especially in the context of plagiarism and code detection.
The tree-based and graph-based approaches have better accuracy than the other approach categories. Especially, if, for example, not enough code documentation and comments are present, text- and token-based approaches will face serious accuracy problems. It seems that there is a recent trend to encode graph-based structures in an appropriate format that is amenable to deep learning. This includes graph kernels [27], graph summaries like structural attentions [28] and graph embeddings [29]. In result, a graph-based classification model [30] can be deduced that is ultrafast to support the accurate, graph-based classification of open source software components.

5.2.2 Automatic class construction

The above constructions assume that the features obtained from the code projects in the training set have been manually classified or *labelled* as belonging to given classes, and this allows the classification methods to use some kind of distance metric for an unknown code project to identify the largest similarity with one of the known code classes. However, it cannot be expected that this knowledge is available, in particularly not when code class data is collected automatically from open source repositories; therefore, it is necessary to investigate other methods that can automatically identify the code classes from the available data.
The most famous clustering algorithm assuming *a priori* knowledge of only the number of classes k to be used is the *k-means* algorithm [31] aiming to place the feature vectors into k clusters so that the total *variance* is minimized. The initial allocation is gradually improved until no further improvement is possible, and this allocation will in general be a local minimum as the optimisation problem is again proven to be NP-Hard [32]. Furthermore, the converged solution tends to be sensitive to the choice of the initial allocation [33], and to provide clusters of approximately the same size. The first issue can be overcome by using a more robust version of the *k-means* clustering algorithm [34] [35] that is evaluated to perform well for a wide range of initial clusters [33], and the second issue can be alleviated by using algorithms that partition the samples around *medoids*, i.e., real members of the data set instead of the average value of the clusters [36].
The *k-means* algorithm is a *parametric* method based on the mean and the variance of the sample classes. A non-parametric classification rule was proposed by Fix and Hodges in 1951 known as the *k-nearest neighbors* classification [37]: A new sample should be assigned to the class most heavily represented among its k already classified neighbours. This *voting* procedure requires a concept of a neighbourhood, and any consistent distance metric can be used to find the k nearest neighbours. Furthermore, it is possible to weight the votes of the neighbourhood, and often *affine* weights are used, i.e., the weights sum to unity. The original approach assumes a weight of $1/k$ assigned to the k nearest neighbours, and zero to all others; or one may weight the votes of the k nearest neighbours based on their distance from the new sample according to the distance metric. Alternatively, one may weight the votes to minimise the risk of assigning the new code sample to the wrong class, and such weights have been found by solving the classification problem asymptotically [38].

5.2.3 Machine learning methods for code classification

In the context of a huge code repository like GitHub, where most of the developers create and maintain their repositories, classical code similarity techniques might face performance and scalability problems while they might

not deliver classification results with a suitable accuracy level. To this end, machine-learning-based techniques have come into play, which attempt to find the right balance between performance and accuracy. The relevant approaches can be classified as supervised and unsupervised. Supervised approaches [39]–[40] map existing tags (from a fixed set) or specific categories (from a fixed set) to software components by utilising a supervised ML technique (e.g., bag-of-words, linear regression or event their combination). Unsupervised approaches [41], [42] utilise unsupervised ML techniques like Latent Dirichlet Allocation (LDA) to cluster software projects into potential categories. Please note that some of the aforementioned approaches are able to produce a hierarchical categorisation of the software projects [43]. In addition, extra work has been proposed to produce meaningful names for the classes in a (flat or hierarchical) categorization [44]. In both sub-categories (supervised and unsupervised), the accuracy is quite satisfactory only when both the source-code and a high-level textual description of the software projects exist. In [45], it seems that this issue is solved through the use of word embeddings to construct a neural classification architecture and train it over a large set of informative software projects that come with an adequate high-level textual description. However, that work still needs some improvement as it produces moderate accuracy results when no description is provided for a software project and assigns just one category to a software project only from a fixed set of categories.

5.2.4 Future plans

Overall, extensive work has been conducted in the area of software code similarity and classification. It seems that recent work on the use of topic maps and word embeddings along with deep learning techniques can lead to a better trade-off between classification accuracy and performance. However, as it was pointed out, such work still needs some extension in order to become more suitable for our current task at hand. Thus, what we are actually proposing to perform as research work on software classification is to attempt to extend an existing, promising approach like the one in [45] with the capability to increase its accuracy in specific situations as well as to produce a hierarchical classification of software projects where multiple categories per project can be assigned. Further, using the categories and the code analysis knowledge, we can functionally match and rank software components. However, while two software components can map to the same categories, this does not mean that they deliver the same functionality. This just signifies that these components are functionally similar. Thus, another suggestion would be to rely on a second-level classification or matching, applied after the first one, which would attempt to employ a high-accuracy approach (i.e., tree- or graph-based) in order to further reduce the matching results only to those which are highly precise. This would, of course, potentially reduce the recall but it can certainly increase the accuracy of the classification.
Another direction that can be taken is to use graph homomorphism and inexact matching when the software can be represented as a graph. This can either be a functional ‘call-graph’ or it can be a ‘data and workflow’ graph, like the Directed Acyclic Graphs (DAGs) often found in High Performance Computing (HPC) applications. It is also possible to base this on pre-identified software- or architectural patterns, which are also graph representations of software.
We intend to explore both directions of work by extending existing classification approaches, where necessary, as well as benchmarking the carefully selected approaches according to specific classification scenarios taken from the state-of-the-art as well as from the use-cases of the MORPHEMIC project. As the result, we envision to design and develop a Classifier component which advances the state-of-the-art by providing both scalable, ultra-fast, as well as highly accurate software classification.

6. Conclusion and next steps

This deliverable provides the description and results of the first set of activities on the Application Profiler, performed during the first period of MORPHEMIC (M1-M12).
One of the more ambitious goals of the project is to provide an automated or semi-automatic application deployment process, able to select the best *deployment configuration*, by making use of *different environments and application forms*. A preliminary operation to achieve this goal is to identify the profile of the software application to be deployed. The Application Profiler, introduced in this document, is the component that will perform this task.
One of the basic functionalities of the Application Profiler is code mining, which consists in the research of code, data and metadata of similar applications to define recurrent profiles and deployment models.
The tools that perform the activity of code mining are part of the architecture of the Application Profiler. Namely, the Web Crawler, the Knowledge Base and the Code Analyser. At the time of writing this deliverable, the three tools are in a different deployment status. The Web Crawler has been implemented, the Knowledge Base is partially implemented and the Code Analyser should be designed. Concerning the Web Crawler, it derives from the EU project MARKOS, chosen after an analysis of different crawling tools (e.g., OpenHub, Krugle). The reasons for this choice are the following:

- *open source license;*
- *reusable, scalable, easily adaptable;*
- part of the offered functionalities is the *Knowledge Base* provided by the Web Crawler; a component that has already been designated as important in the Application Profiler architecture.

A further step is to associate to the Web Crawler, the appropriate information where open source projects can be retrieved. Some of them were already associated to the MARKOS' version (such as GitHub), while others have been identified and analysed as candidate repositories (for example CPAN, CTAN). In parallel with the implementation of the remaining modules of MORPHEMIC, the data needed for the definition of application profiles will become clear. This will be one of the selection factors for new repositories among the current candidates. This will be the next step of the work on the Web Crawler. Concerning the Knowledge Base, as mentioned before, it is partially implemented as part of the Web Crawler, specifically the DOAP database.
During the second year of the project, it will become an autonomous RESTful service, completely separated from the crawling process and accessible by all the other components. In this sense, for example, the Classifier will be able to exploit the metadata stored in the Knowledge Base to create code classifications; the Camel Designer will leverage these data to allow users to analyse the functionality of the application components and the open-source components matching them; the Profile Maintainer will provide users with data from the Knowledge Base to check if new configurations have been discovered for a specific application.
Code classification is another important research activity performed in this context. In fact, code mining identifies various types of projects that can be considered as the source for code classes, such as High-Performance Computing (HPC) code or web code.
This code should be classified utilising appropriate methods to enable the retrieval of optimal deployment models. Several characteristics are taken into consideration for the classification:

- *qualitative characteristics*, such as the programming languages used;
- *structural features* reflected in the application call graph, i.e., how components and functions invoke other components and functions;
- *the data structures in use*, e.g., fixed-size arrays or dynamically scalable vectors or lists.

Other important factors to consider are the data processing software and the data processing graph in which some components process data before others and which will also reveal the type of application.

The result of the code classification research will be the realization of the Application Profiler component identified as Classifier.
In the first period of the project, three steps have been identified to successfully classify an application code base:

- *feature definition*;
- *measurements*;
- *classification.*

The feature definition is the first step to identify common features between the various types of code, it should be able to:

- *differentiate* among classes;
- *be available* within all code types.

In general, the process of feature definition takes into account different kinds of features:

- *ordinal,* such as a programming language or a used data structure;
- *quantitative,* such as lines of code, number of functions, classes or components;
- *functional,* such as the software models identified within the code.

A complete feature definition process has not been performed yet. Indeed, the process should be fully defined according to the characteristics of the MORPHEMIC code mining. The major challenge for the next project period is to tune the generic process on these characteristics, and, after that, to measure them automatically. The code

classification will complete the code mining process. This work fully defines the functionality of code mining. Part of the tools have been already implemented, part of them will be released in the second year of the project. They represent one of the core functionalities on which the Application Profiler will be built on. The other ones will be described in the next deliverables of WP3, specifically in the deliverable D3.2 “Automatic source code identification of deployment modules” (M32).

In this context, an important aspect of future work on application profiling is studying and analysing application profiling for hardware acceleration. For instance, in the case of accelerators, the main difference between CPU and GPUs or FPGAs is that the application needs to be translated to a domain-specific language (e.g., CUDA for GPUs and OpenCL for FPGAs). One research direction will be to understand how the translation of some functions to the domain specific languages can be done in advance and not at runtime. The idea could be to develop a library of the most widely used functions (e.g., compression, encryption, etc.) using the domain-specific languages available for the acceleration platforms in MORPHEMIC. In order to utilize these accelerators, we need to adapt the profiling of the applications to recognize which of these functions are available as accelerators. In that case, these applications can be labelled as such and be executed also on accelerators. For example, the profiling of the application must search for specific keyworks indicating some type of compression. If the function, GZIP () for example is used, then this application can be labelled as “accelerator” meaning that this function could be offloaded to the accelerators since there is an available design in GPU/FPGA that accelerates the GZIP function execution.

7. References

[1] A. Ranganathan, C. Shankar, and R. Campbell, "Application polymorphism for autonomic ubiquitous computing," *Multiagent Grid Syst.*, vol. 1, no. 2, pp. 109–129, Jan. 2005, doi: 10.3233/MGS-2005-1205.

[2] H. Liu and H. Motoda, *Feature Extraction, Construction and Selection: A Data Mining Perspective*. Springer Science & Business Media, 1998.

[3] S.-H. Chen, *Big Data in Computational Social Science and Humanities*. Springer, 2018.

[4] Y. Verginadis, I. Patiniotakis, and G. Mentzas, "Metadata Schema for Data-Aware Multi-Cloud Computing," in *2018 Innovations in Intelligent Systems and Applications (INISTA)*, Thessaloniki, Jul. 2018, pp. 1–9, doi: 10.1109/INISTA.2018.8466270.

[5] G. Bavota *et al.*, "The market for open source: An intelligent virtual open source marketplace," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, Feb. 2014, pp. 399–402, doi: 10.1109/CSMR-WCRE.2014.6747204.

[6] "Gosain, A., & Sharma, G. (2015). Static analysis: A survey of techniques and tools. In Intelligent Computing and Applica." .

[7] "Cesare, S., & Xiang, Y. (2012). Software similarity and classification. Springer Science & Business Media." .

[8] "Esmaeilpour, M., Naderifar, V., & Shukur, Z. (2014). Design pattern mining using distributed learning automata and DNA s." .

[9] "Rouchka, E. C. (2006). Aligning DNA sequences using dynamic programming. XRDS: Crossroads, The ACM Magazine for Students." .

[10] "L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison wit." .

[11] "C. K. Roy and J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and." .

[12] "K. Tate, Sustainable software development: an agile perspective. Upper Saddle River, NJ: Addison-Wesley, 2006." .

[13] "S. Schleimer, D. S. Wilkerson, and A. Aiken, 'Winnowing: local algorithms for document fingerprinting,' in Proceedings o." .

[14] "H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, 'SourcererCC: scaling code clone detection to big-code,.'" .

[15] "L. Jiang, G. Misherghi, Z. Su, and S. Glondu, 'DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones,' in 2." .

[16] "C. Ragkhitwetsagul, J. Krinke, and D. Clark, 'A comparison of code similarity analysers,' Empir. Softw. Eng., vol. 23, n." .

[17] "J. A. W. Faidhi and S. K. Robinson, "An empirical approach for detecting program similarity and plagiarism within a univ." .

[18] "Cory Capser and Michael W. Godfrey, 'Toward a taxonomy of clones in source code: a case study,' Amsterdam, The Netherlan." .

[19] Michel Marie Deza and Elena Deza, *Encyclopedia of Distances*, 4th ed. Berlin, Heidelberg: Springer Berlin / Heidelberg, Springer Berlin Heidelberg, Springer, 2016.

[20] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat, "NP-hardness of Euclidean sum-of-squares clustering," *Mach. Learn.*, vol. 75, no. 2, pp. 245–248, May 2009, doi: 10.1007/s10994-009-5103-0.

[21] "I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, 'Clone detection using abstract syntax trees,' in Proceedin." .

[22] Peter Weiner, "Linear pattern matching algorithms," in *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, Conference Location: USA, Oct. 1973, pp. 1–11, doi: 10.1109/SWAT.1973.13.

[23] Martin Farach, "Optimal suffix tree construction with large alphabets," in *Proceedings 38th Annual Symposium on Foundations of Computer Science*, Conference Location: Miami Beach, FL, USA, Oct. 1997, pp. 137–143, doi: 10.1109/SFCS.1997.646102.

[24] Paul Jaccard, "Distribution de la flore alpine dans le Bassin des Dranses et dans quelques régions voisines," *Bull. Société Vaudoise Sci. Nat.*, vol. 37, no. 140, pp. 241–272, 1901, doi: 10.5169/seals-266440.

[25] "D.-K. Chae, J. Ha, S.-W. Kim, B. Kang, and E. G. Im, 'Software plagiarism detection: a graph-based approach,' in Proceed." .

[26] "K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on And." .

[27] “N. M. Kriege, F. D. Johansson, and C. Morris, ‘A survey on graph kernels,’ Appl. Netw. Sci., vol. 5, no. 1, p. 6, Dec. 2.” .
[28] “J. B. Lee, R. Rossi, and X. Kong, ‘Graph Classification using Structural Attention,’ in Proceedings of the 24th ACM SIGK.” .
[29] “P. Goyal and E. Ferrara, ‘Graph embedding techniques, applications, and performance: A survey,’ Knowl.-Based Syst., vol.” .
[30] “C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, “ProGraML: Graph-based Deep Learning for Program Opti.” .
[31] Stuart P. Lloyd, “Least squares quantization in PCM,” *IEEE Trans. Inf. Theory*, vol. 28, no. 2, pp. 129–137, Mar. 1982, doi: 10.1109/TIT.1982.1056489.
[32] M. R. Garey, D. S. Johnson, and Hans S. Witsenhausen, “The complexity of the generalized Lloyd - Max problem (Corresp.),” *IEEE Trans. Inf. Theory*, vol. 28, no. 2, pp. 255–256, Mar. 1982, doi: 10.1109/TIT.1982.1056488.
[33] M. Emre Celebi, Hassan A. Kingravi, and Patricio A. Vela, “A comparative study of efficient initialization methods for the k-means clustering algorithm,” *Expert Syst. Appl.*, vol. 40, no. 1, pp. 200–210, Jan. 2013, doi: 10.1016/j.eswa.2012.07.021.
[34] David Arthur and Sergei Vassilvitskii, “k-means++: the advantages of careful seeding,” in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, Conference Location: New Orleans, LA, USA, Jan. 2007, pp. 1027–1035, doi: 10.5555/1283383.1283494.
[35] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii, “Scalable k-means++,” *Proc. VLDB Endow.*, vol. 5, no. 7, pp. 622–633, Mar. 2012, doi: 10.14778/2180912.2180915.
[36] Erich Schubert and Peter J. Rousseeuw, “Faster k-Medoids Clustering: Improving the PAM, CLARA, and CLARANS Algorithms,” in *Proceedings of the 12th International Conference on Similarity Search and Applications (SISAP 2019)*, Conference Location: Newark, NJ, USA, Oct. 2019, vol. 11807, pp. 171–187, doi: 10.1007/978-3-030-32047-8_16.
[37] Evelyn Fix and J. L. Hodges, Jr., “Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties,” *Int. Stat. Rev.*, vol. 57, no. 3, pp. 238–247, 1989, doi: 10.2307/1403797.
[38] Richard J. Samworth, “Optimal weighted nearest neighbour classifiers,” *Ann. Stat.*, vol. 40, no. 5, pp. 2733–2763, Oct. 2012, doi: 10.1214/12-AOS1049.
[39] A. Sharma, F. Thung, P. S. Kochhar, A. Sulistya, and D. Lo, “Cataloging GitHub Repositories,” in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering - EASE’17*, Karlskrona, Sweden, 2017, pp. 314–319, doi: 10.1145/3084226.3084287.
[40] “Y. Kim, S. Cho, S. Han, and I. You, “A software classification scheme using binary-level characteristics for efficient s.” .
[41] “K. Tian, M. Revelle, and D. Poshyvanyk, ‘Using Latent Dirichlet Allocation for automatic categorization of software,’ in.” .
[42] “X. Cai, J. Zhu, B. Shen, and Y. Chen, ‘GRETA: Graph-Based Tag Assignment for GitHub Repositories,’ in 2016 IEEE 40th Ann.” .
[43] “T. Wang, H. Wang, G. Yin, C. X. Ling, X. Li, and P. Zou, “Mining Software Profile across Multiple Repositories for Hiera.” .
[44] “A. Hindle, N. A. Ernst, M. W. Godfrey, and J. Mylopoulos, “Automated topic naming to support cross-project analysis of s.” .
[45] “A. LeClair, Z. Eberhart, and C. McMillan, ‘Adapting Neural Text Classification for Improved Software Categorization,’ in.” .