Automatic source code identification of deployment modules

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 –
30 June 2023

www.morphemic.cloud

Deliverable reference
D3.2

Date
31 March 2023

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 the results of the Application Profiling activities; in particular it describes how the Code Classifier supports the automatic building of the initial deployment model and its impact on the automatic building process based on feedback from application developers and comparison with manually constructed deployment models and the Profiler as result of the implementation of Application Profiling feature as:

- Code mining methodologies and technologies from which metadata are retrieved and stored (the corresponding MORPHEMIC's [1] components are the Crawler and the
- The deployment modelling to find the initial deployment (the feature is implemented by the CAMEL Designer [2] integrated with the Profiler).
- The classification technologies and algorithms for classifying open-source projects that can be considered as the source for code classes, such as High-Performance Computing (HPC) code or web code.

The Profiler interoperates with:

- CAMEL Designer to find the initial CAMEL model [2].
- Crawler and KnowledgeBase to search and find the metadata of open-source projects.
- Classifier to find the best application deployment model.

Code classification according to project metadata supports and enhances the definition of the Application Profile, being used to obtain the best possible adaptation of the application deployment to suitable infrastructures and component configurations. The Profiler works by comparing the application against a certain number of profiles pre-defined in the MORPHEMIC [1] platform. Among them the most suitable one is selected, and its deployment model applied. In the first part of the MORPHEMIC [1] project components have been designed and developed to apply techniques (e.g., code mining and crawling) to retrieve and store projects' metadata information. The second part of the project has been more focused on classification techniques for the open-source projects in order to support the CAMEL Designer tool [2] (a component of MORPHEMIC [4]) for the initial deployment CAMEL model [2] that helps to define the profile of an application at different level: instance, type, configuration (for application components such as CPU, GPU) and communication with the other components of the MORPHEMIC [1] platform.

Author(s)

Amir Taherkordi (UiO), Ciro Formisano (ENG), Dapeng Lan (UiO), Jean-Didier Totow (UPRC), Maria Antonietta Di Girolamo (ENG).

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
10-01-2023	1.0	ENG	1st draft version
13-03-2023	1.1	UiO	2nd draft version: Chapter 4
27-03-2023	1.2	ENG	3rd draft version: Revision of the document structure: figure, table, indentation, bibliography.
03-04-2023	1.3	UPRC, ENG	4th draft version: bag of words section, conclusion, table and figure.
04-04-2023	1.4	UiO	Conclusion
04-04-2023	2.0	UiO, ENG, UPRC	Ready for the internal review
07-04-2023	2.0	ICCS	1st internal review
11-04-2023	2.0	ICON	2nd internal review
12-04-2023	2.1	ENG,UPRC,UiO	Implementation of the reviewers' comments, feedbacks
20-04-2023	3.0	ENG	Updating figures, comments
26-04-2023	3.1	UPRC	Implementation of the reviewers' comment
27-04-2023	final	ENG	Final version

Table of Contents

Revisions 2
Index of Figures 4
Index of Tables 5
Glossary 6
1. Introduction 7
1.1 Scope 7
1.2 Intended Audience 8
1.3 Document Organization 8
2. Application Profiling: Profiler 9
2.1 Introduction 9
2.2 Profiler 10
2.3 Profiler Implementation 11
2.4 Profiler interactions with the other application profiling components 13
2.4.1 CAMEL Tools 14
2.4.2 Code mining components 14
3. Code classification results: Classifier 19
3.1 Introduction 19
3.2 Results about the feature definition process 20
3.2.1 Evaluation of the code classification tools 21
3.3 Classifier 22
3.3.1 Internal Architecture 22
3.3.2 Functional Analysis 22
3.3.3 Technical Analysis 23
3.4 Evaluation of the Code Classification Tool 23
3.4.1 LLVM 24
3.4.2 Joern 24
3.5 Build the hierarchical graph 26
3.5.1 Inst2vec 26
3.5.2 Combining Inst2vec and RNN 26
3.6 Results 27
3.7 Classification results in BoW 29
3.8 Evaluation metric for classification models 30
4. Conclusion 32
5. References 33

Index of Figures

Figure 1: Polymorphic adaptation architecture....7
Figure 2 Application Profiling components....9
Figure 3 Application Profiling implementation....10
Figure 4 Profiler Architecture....11
Figure 5 Interaction of the Application Profiling components....13
Figure 6 How to work the Crawler....16
Figure 7 Time Query Data....16
Figure 8 KnowledgeBase data schema....17
Figure 9 The sequence diagram of the KnowledgeBase....18
Figure 10 Different forms of representing code using graphs....20
Figure 11 Dependencies between program representations....20
Figure 12 Feature extraction from the code property graph....21
Figure 13 Code Classifier design....22
Figure 14 The pipeline for code classification by graph neural network....23
Figure 15 Three phase design three-phase compiler....24
Figure 16 Code example for Joern....25
Figure 17 Embed nodes within other nodes and to cluster nodes together hierarchically....26
Figure 18 Data label code snippets....27

Index of Tables

Table 1 The changes of the KnowledgeBase 15
Table 2 Evaluation of the code classification tools 21
Table 3 Test results 29
Table 4 Evaluation results 31

Glossary

ABBREVIATIONS	
AI	Artificial Intelligence
API	Application Programming Interface
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CP	Constraint Problem
CRAN	Comprehensive R Archive Network
CTAN	Comprehensive TeX Archive Network
CUDA	Compute Unified Device Architecture
DNA	DeoxyriboNucleic Acid
DOAP	Description of A Project
DT	Dominator Tree
ETL	Extract Transform Load
EMS	Event Management System
FPGA	Field Programmable Gate Array
HPC	High Performance Computing
GNN	Graph Neural Network
GPU	Graphics Processing Unit
GZIP	Gnu Zip
JSON	JavaScript Object Notation
LCS	Longest Common Subsequence
LDA	Latent Dirichlet Allocation
BOW	Bag Of Words
MDS	Metadata Schema
ML	Machine Learning
MPL	Mozilla Public License
N/A	Not Available
OSP	Open-Source Project
PCA	Principal Component Analysis
PDG	Program Dependence Graph
PDT	Post Dominator Tree
PMC	Project Management Committee
RDF	Resource Description Framework
RNA	RiboNucleic Acid
REST	Representational State Transfer Application Programming Interface
SOTA	State-Of-The-Art
SQL	Structured Query Language
TFDIF	Term Frequency–Inverse Document Frequency
UIUC	University of Illinois Urbana-Champaign
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 capability of automatic building (of the MORPHEMIC[1] platform) an initial optimal deployment model based on the code classifier and the feedback provided from the application developers and the result of the comparison with manually constructed deployment models.

The task is part of the Polymorphic Adaptation activities which aim at allowing applications to be deployed in different environments (as multi-cloud, edge, fog) and change their configurations based on the application features and context in order to maximize the application performance. In the first part of the MORPHEMIC project a deeper analysis on what are the best methods and tools for the definition of the Application Profiler is provided as part of the deliverable "D3.1-Software tools and repositories for the code mining" [3]. Another important result is to design and implement the Polymorphic Adaptation feature and architecture (the last version is provided in the deliverable "D3.4 Planning and adaptation results" [10]) as showed in the Figure 1: Polymorphic adaptation architecture:

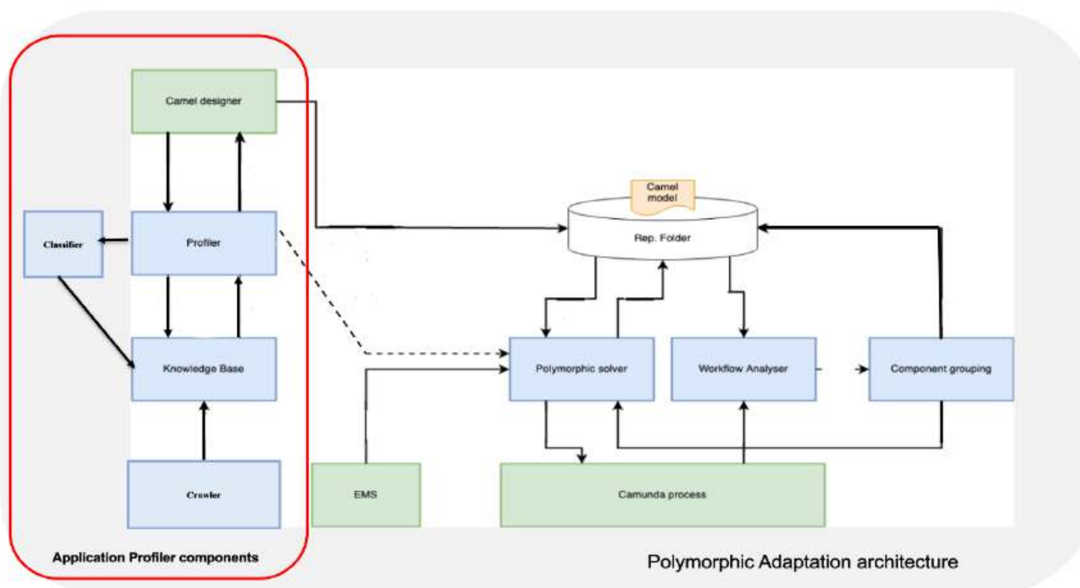

Figure 1: Polymorphic adaptation architecture

The second part of the MORPHEMIC project is focused on the implementation of the modules of the Application Profiler (Figure 1: Polymorphic adaptation architecture, rounded rectangle red), in particular the implementation of the module which models and classifies the applications retrieved and stored by the code mining tools [5].
Specifically, the Profiler provides three specific features:

- Code mining tools: Crawler and KnowledgeBase [3].
- Deployment modelling: Camel Model for supporting polymorphic application modelling and subsequently management as provided in the deliverable "D1.1 Data, Cloud Application & Resource Modelling and Final Data, Cloud Application & Resource Modelling."[6].
- Code classification tools as described in the paragraph 3.2.1 "Evaluation of the code classification".

The Web has been considered the best place where information to build application profiles can be retrieved. Specifically, the Web provides information on open-source projects, from which the pieces of information closest to the application to be deployed should be selected. For this purpose, code mining techniques are applied in MORPHEMIC to extract specific information from specific repositories: the information is mainly represented by metadata, such as service endpoint, release number, topic, label [5].

In addition, a pre-requirement for getting all the benefits provided by MORPHEMIC is to model the application to be deployed as a *polymorphic application*[6]. Domains and metadata related to multi-cloud should be covered. Such domains include the application's deployment topology, its own requirements as well as its measurement. While the management goals and activities include not only application monitoring but also reactive and proactive application adaptation. In order to implement this, Morphemic uses some features of the MELODIC [4] project: CAMEL modelling language [2] and the metadata schema (MDS) [7].

CAMEL [2] is already a rich cloud application modelling language; in MORPHEMIC it has been extended by including elements to model polymorphic applications (as label, category) and MDS such as the coverage of all kinds of component configurations and the mapping of those configurations to corresponding requirements. CAMEL Models can be produced by using the CAMEL Designer tool which is a module of Modelio modelling [8].

Another important aspect is the code classification that completes the code mining process for the definition of the application profiling.

In the first project period of the MORPHEMIC project, different types of open-source projects have been identified through code mining tools providing classifications such as HPC or web code [5]. This activity has been performed by using another important component involved in the application profiling process: the Classifier.
The code is classified:

- utilizing specific methods to retrieve optimal deployment models (as qualitative characteristics, structural features, the used data structures) and
- identifying different features as ordinal, quantitative and functional.

In this context, an important result is to define application profiles based on the analysis of application developed for hardware acceleration. To this end, the differences of applications developed for GPUs or FPGAs are particularly important. Specifically, each hardware accelerator needs a specific domain language (e.g., CUDA for GPUs and OpenCL for FPGAs). This specific feature identifies the features of the application enabling them to be classified, labelled (CUDA, FPGA, GPU) and adapted to be executed on the specific accelerator. For example, if the profiling of the application must search for specific keyworks indicating some type of compression, 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 [3].

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, 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 [9].

1.3 Document Organization

The current chapter introduces the scope, the objectives, and the structure of this document.
The remaining chapters are the following:

- Chapter 2 (“Application Profiling: Profiler”): the focus is on the description and implementation of the Profiler and its components. Reference to the MORPHEMIC deliverables:
    - “D3.1-Software tools and repositories for the code mining” [3] are provided for the description of the architecture and design of the Profiler and what is changed in its components for the code mining tools (Crawler and KnowledgeBase).
    - “D5.1 User Interfaces Specification” [8] and “D1.1 Data, Cloud Application & Resource Modelling” [6] are provided for the description of the use and architecture of the CAMEL tools (Designer and Model).
    - “D3.4 Planning and adaptation results” [10] are provided for the interaction and information about the used communication protocol, the sequence diagram of the messages exchanged among the Profiler and the code mining, code classifier and camel tools.
- Chapter 3 (“Code classification results: Classifier”) describes the results of the analysis of the tools and algorithms selected and listed in the deliverable “D3.1-Software tools and repositories for the code mining” [3].
- Chapter 4 (“Conclusion") provides the final considerations and planned next steps.

2. Application Profiling: Profiler

2.1 Introduction

The Application Profiling is one of the main tasks of the Polymorphic Adaptation feature [10] that helps the deployment and the reconfiguration of the polymorphic applications that change their architecture at runtime by selecting a different application component form from those possible based on their requirements and context[3].

The Profiler is the implementation of the analysis of the functionality for the Application Profiling task conducted in the first part of the MORPHEMIC project [3].

Many components are involved in the definition of the best application profile in order to define the best deployment model for a user’s application. Figure 2 shows the final architecture of the Application Profiling components where the colours denote different components:

- **Blue** denotes new components of the Profiler.
- **Red** denotes other Melodic/Morphemic platform components.
- **Grey** denotes various kinds of information/knowledge sources (e.g., benchmarking sites and open-source software repositories).
- **Orange** denotes model-based artefacts that are exploited by the Profiler.

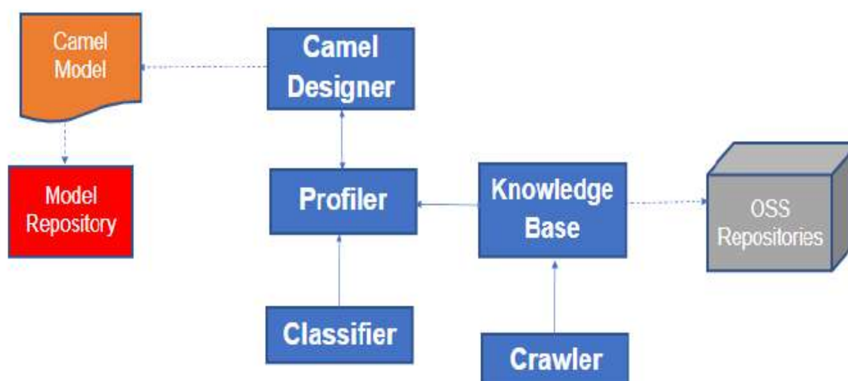

Figure 2 Application Profiling components

The following section describes the details about the role and technical details of the Profiler.

2.2 Profiler

The main responsibility of the Profiler is to construct, enhance and maintain application profiles covering both the functional and non-functional aspects as well as all the application forms [11]. The capability to cover non-functional aspects enables to produce continuously improved performance models for application components, from which more precise and optimal resource selection improves adaptive application deployment.

Another virtue of the Profiler is that it can deal with multiple applications and that it carries classification knowledge that can be beneficial for multiple application kinds. Discovering possible application forms if not explicitly defined by the application owner is a crucial task for enabling polymorphism on a cloud management platform. This procedure requires advanced technics of crawling, feature extraction and classification for establishing the application profile that can be exploited by a cloud management platform. In other terms, the profiling can be considered as a technique of application components recognition.

MORPHEMIC's approach consists of targeting (filtering) relevant open-source projects from different repositories (GitHub[1]) or open-forge (R-forge[2]) or a list of directories (Apache[3]). By relevant repositories, we mean application (component) containing a label corresponding to a relevant cloud application form such as serverless, GPU, FPGA, HPC etc. The selection of a relevant project could be based on a label present to project metadata or the presence of a library in the source code leading to an interested form. For instance, the usage of CUDA or XILINX libraries lead to respectively GPU and FPGA. The presence of Docker file points to a docker container form. After having downloaded these repositories, we will then proceed to the feature extraction where the bag of words (BoW) technique is used. The BoW engine will collect words inside the source code, we are interested in imported libraries to reduce the feature size collection. Therefore, for each project, an object is created containing the project's name, project's labels, programming language and a collection of words. The object will be stored in the database.

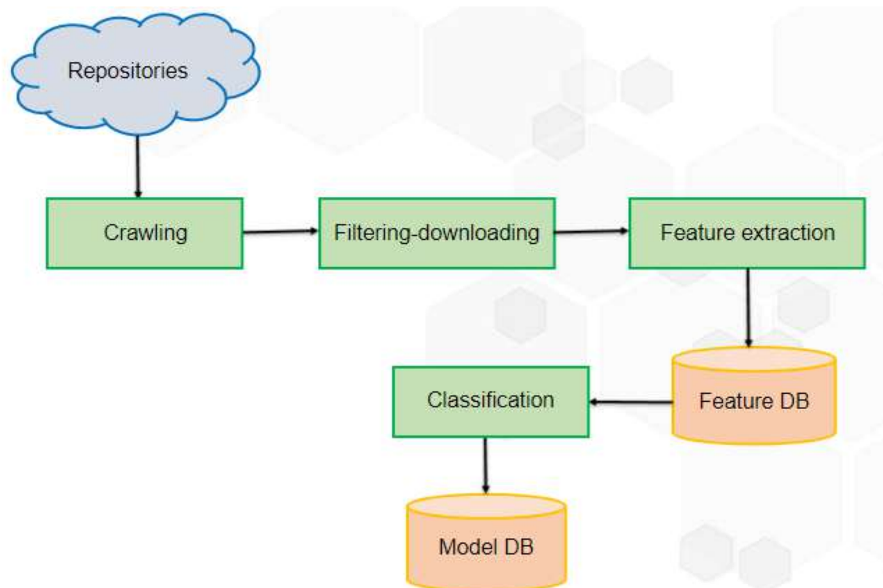

Figure 3 Application Profiling implementation

The result of the implementation (Figure 3) of this architecture is to implement as first step the capability to crawl open-source repositories from forges and meta-forges and analyse their code in or der to detect new forms for application components [12]. This detection is quite important in the context of polymorphic computing as:

[1] https://github.com/
[2] https://r-forge.r-project.org/
[3] https://www.apache.org/

- it reduces the burden to develop multiple application component forms to support true polymorphism.
- it can suggest new application component forms when the existing ones are insufficient to cope with all possible application situations.
- also supplies the capability to generate and maintain application profiles, i.e., characterisations of the application functionality and performance for the different existing application forms.

Another important role is covered by the classification technologies to define an application profiling. In this regard, in the deliverable "D3.1-Software tools and repositories for the code mining"[3], various tools were analyzed, with particular attention to the classification of applications based on specific bag-words modelled into the CAMEL Designer model such as FPGA, GPU, docker, Kubernetes.

2.3 Profiler Implementation

The result of the Application Profiling task is the Application Profiler, called, from now on Profiler, to improve the readability of the document. Specifically, the Profiler consists of:

- **Repository Filter**: for deciding whether a repository based on its metadata should be analysed or not [11].
- **Downloader**: for downloading the repository code [11].
- **Orchestrator**: for the Configuration and BoW extractor components.
- **Extractor:** Profiler has an analyser manager that receives from extractors components (as Configuration Extractor, Quality Extractor, Bow Extractor, Graph Extractor, Licence Extractor) different features (analysed from each analysis manager extractor and extracted from the same extractors, as showed in Figure 4):

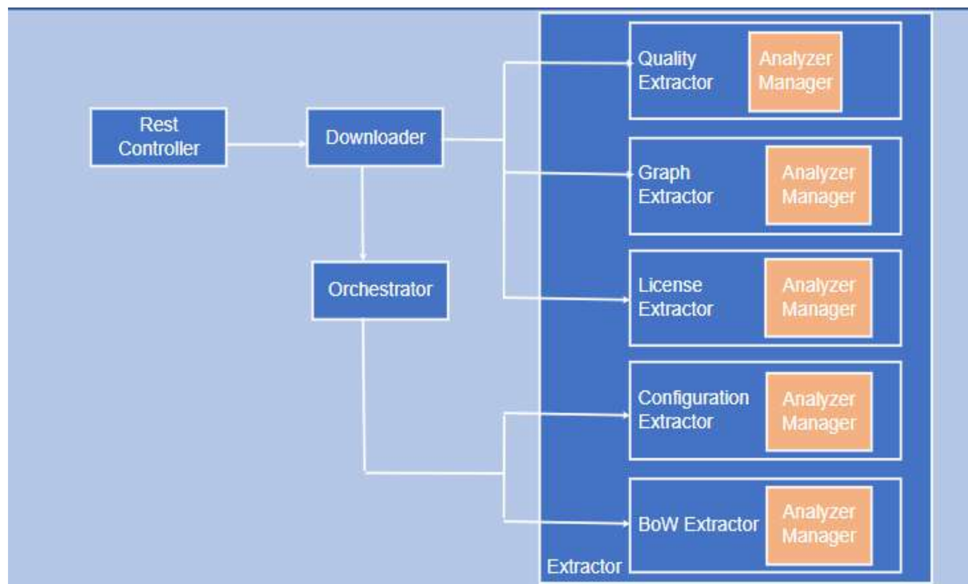

Figure 4 Profiler Architecture

Each analyser implements a method called *"addProject"* with the input parameter:

- **Repo object**: containing information about the repository where open-source projects metadata are retrieved, specifically:
    - "name": name of the repository.
    - "url": the full path of the repository.

Another method implemented by each analyser is the *"setOuput"*, called after the analysis task has been completed with three input parameters:

- **Method**: the name of the analyser (for example License Extractor, Repository Filter)
- **Data**: a dictionary, the analyser output

- **Repoobject**: contains the name of the repository. The repo object received from the analyser manager by the method “*addProject”*.

The description of the role of each component is provided in the following subsections:

Repository Filter

The Repository Filter is responsible for polling the KnowledgeBase to retrieve newly generated or updated repository metadata obtained from the Crawler. According to filtering criteria provided by the DevOps user, for each repository (metadata), the component checks whether it should be included in the analysis and classification tasks. Accepted repositories are forwarded to the Orchestrator to perform the remaining analysis tasks. The filtering criteria are advocated to rely on parts of the metadata like the programming language, the related topics/categories, and the repository type (i.e., public or private).

Downloader

Once a repository is eligible for further processing, its latest release URL is passed to the Downloader, a component responsible for downloading the release and storing it in a proper place of the local file system. Once this task is accomplished, the Orchestrator is informed to invoke the feature extraction task. All dedicated components to the feature extraction task are finally executed in parallel by the Orchestrator. Each of these components inspects the latest repository release and extracts features of a specific kind based on it. These features are then stored back in the KnowledgeBase at the appropriate place (related to the current repository).

Configuration Extractor

The Configuration Extractor extracts configuration classes as features, like FPGA, GPU, container & serverless.

License Extractor

The License Extractor is responsible for extracting license's information from each source code information sent to the KnowledgeBase as the licensing statement reported in the file, its contributors, the licenses specified in it, the control copyright years. This can be considered as an additional, optional filtering functionality for the data repositories information open-source projects provided from the Profiler Downloader component.

BoW Extractor

The Bag of Words (BoW) technique can be used for code static analysis by treating the source code as a document and each source code file as a collection of documents. This allows us to represent each file as a vector of word frequencies and apply machine learning techniques for tasks such as code classification and clustering. In our case the collection involves the concatenation of all the files of a single repository, since we want to classify repositories.
To use BoW for code static analysis, we first need to pre-process the source code. In our context this consists of removing everything except from the included libraries. We will also keep in our collection the name of configuration files. Once the source code has been pre-processed, we can create a vocabulary of unique words in the corpus. We can then represent each source code file as a vector of word frequencies, where each dimension corresponds to a unique word in the vocabulary and the value corresponds to the frequency of that word in the collection.
With the vector representation of all source code files of a repository (input vector), we then apply clustering to group similar repository together or to identify projects that exhibit certain patterns or properties. The machine operation consist of the correlation between the input vector representing the repository with label corresponding to the output vector defining the repository category. As illustration, for a repository containing “main.py” and Dockerfile identified with labels “GPU” and DOCKER. If the main.py has the following content:

```
import cuda
from datetime import datetime
Import pandas as pd
…
…
```

The input vector will be constructed with [“cuda”, “datetime”, “pandas”, “dockerfile”] array since “import, from, as” are considered as stop words. If the supported categories are [“GPU”, “FPGA”, “DOCKER”, “SERVERLESS”, “HPC”] the out vector corresponding to “GPU” and “DOCKER” will be 1 0 1 0 0. We are using a Sequential neural network for the correlation between the input vector and the output vector.

Quality Extractor

The Quality Extractor scans all files related to programming code and extracts code quality-related features.

Graph Extractor

It creates a code graph, which could take different forms (e.g., data flow graph, control flow graph, intermediate program representations, etc). More details are provided in the paragraph 3.3 Classifier.

Rest Controller

The REST Controller is the component that exposes a REST interface for the Profiler, which will enable its proper integration with the MORPHEMIC's Control Plane. Such an interface could be exploited by other profiler components (i.e., exatractors components) in order to request the analysis of specific repositories (e.g., those that map to application components) [10].

2.4 Profiler interactions with the other application profiling components

The Profiler works offline and interoperates with the other tools involved in the definition of the application profiling as:

- The code mining tools (Crawler and KnowledgeBase) in order to retrieve and store the correct metadata information from the application open-source project.
- The deployment tools (CAMEL Designer) in order to find the initial CAMEL Model [13].
- Code classification tools as Classifier in order to classify the best application open-source project.

The following figure provides the Application Profiling tools and how the Profiler communicates with them:

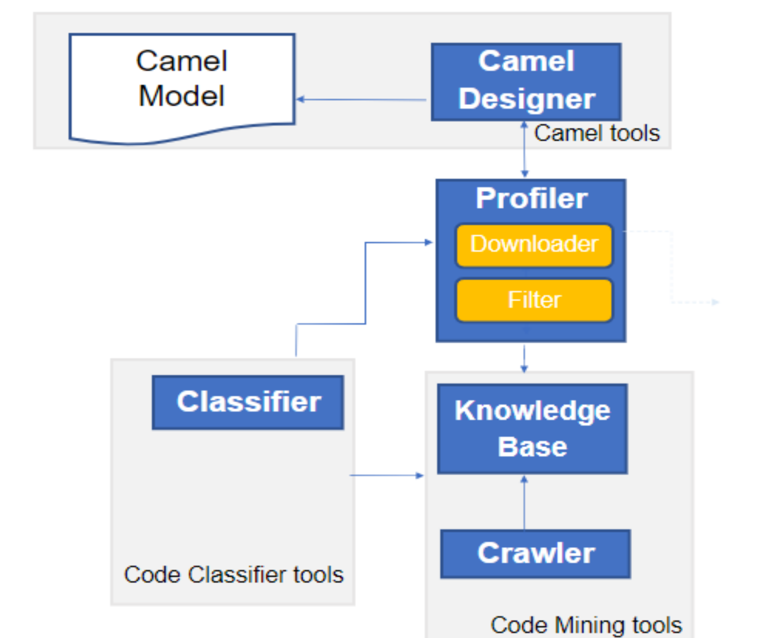

Figure 5 Interaction of the Application Profiling components

Profiler interacts directly with:

- **Code Mining tools** as Crawler and KnowledgeBase to search for open-source projects metadata by crawling the web and store them.
- **Camel tools** as CAMEL Designer as initial deployment model based on the Camel Model as described in the MORPHEMIC deliverable "D5.2 User Interface Guidelines" [13].
- **Code Classification tools** as analysed in the MORPHEMIC deliverable "D3.1-Software tools and repositories for the code mining" [3] and evaluated and implemented as described in the paragraph 3.3 Classifier.

The details about the interaction between the Profiler and these tools will be described in the section about the Polymorphic Adaptation processes of the deliverable D3.4 - "Planning and adaptation results" [10]. Information will be provided about the used communication protocol, the sequence diagram of the messages exchanged among the Profiler and the Code Mining, Code Classifier and CAMEL Tools.

The following section updates the description of the tools directly working with the Profiler as the Code Mining tools [3] and CAMEL Designer [13].

2.4.1 CAMEL Tools

The CAMEL tools allow the application modelling process (using CAMEL Model [8]) to select the best model for a certain application. Among CAMEL tools, the CAMEL Designer plays the main role on this task. Specifically, it helps:

- to implement the profile of an application at distinct levels: instance, type, configuration (for the application components as CPU, GPU for example) and communication with the other components of the MORPHEMIC platform.
- to design the best possible deployment model according to the specifications of the initial deployment model as provided by the CAMEL Model.

In the following section we provide:

- a short description of the component.
- the description of the Profiler that communicates with the CAMEL Designer.

CAMEL designer

The CAMEL Designer is an Open-Source extension of Modelio Modelling Tool for conceiving CAMEL Model and all its sub models and components in an ergonomic, intuitive, and user-friendly environment[13].

The CAMEL Designer is responsible for:

- extending the MORPHEMIC project vision through the simplification of the Cloud application modelling to take advantage of beneficial Cloud capabilities.
- providing an user friendly interface for the user's application modelling.

More technical details about this component are provided in the deliverable "D5.2 User Interface Guidelines - Chapter 3 – CAMEL Designer: A Modelio Environment for CAMEL Modelling Requirements for Morphemic UI features"[13]. The details about the interaction between CAMEL Designer and Profiler (communication protocol, sequence diagram of data) is described in the polymorphic adaptation processes will be described in the MORPHEMIC deliverable D3.4 - "Planning and adaptation results" [10].

2.4.2 Code mining components

An important role is conducted by the code mining tools to help the Profiler to enrich the functional application profile with new configuration suggestions and construct an initial performance model of the application[3]. The first prototype of the Crawler for the MORPHEMIC platform was identified in the first year of the project. This component included also the functionalities of the KnowledgeBase. In the second year of the project some changes have been introduced:

- For greater clarity and readability, the name of the metadata information retrieved from the Crawler repositories configured as "repositories metadata information".

- Crawler's architecture has been revised to optimize some functionalities, such as repositories search and metadata store. Specifically, the changes involving the data storage improved the capability to store structured and non-structured data in real-time.

Based on this last important change, the code mining tools provide two components:

- **Crawler**: responsible for the crawling and parsing of the repositories metadata information retrieved from the dedicated configured open-source repositories (GitHub, Apache, and so on).
- **KnowledgeBase**: responsible for the storage of the repositories metadata information. It is also responsible of sending this information to the Profiler.

The new prototype of Crawler and KnowledgeBase is described in the next sections.

Crawler

The crawling functionality can be considered as the first step of the classifying approach used to define the application profile to improve the application deployment model.
The role of the Crawler is to scrape and find data source-code information of the open-source project from specific and configured open-source repositories (these data are called repositories metadata information in the MORPHEMIC project) such as GitHub, Apache or jQuery Plugin, C-tan, R-forge and Eclipse [3]). These repositories metadata information at the end of this process are classified by the Profiler and are made available to other MORPHEMIC components to support the Polymorphic Adaptation Task.

In the first part of the MORPHEMIC project, the first prototype of the Crawler provided storage functionalities too as [12]:

- **maintain** a list of data sources and structured information related to open-source projects.
- **download** automatically and continuously part of that information.
- **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.

However, given the large amount of data to be analyzed (coming from the crawler repositories), a specific component has been designed and deployed to provide storage functionalities. This component is the KnowledgeBase.

The result of this new design is that the Crawler now plays only the role of data scraping and parsing, and the KnowledgeBase is a separate service that stores the data parsed by the crawler and send them to the Profiler. The modification did not impact the architecture of the Crawler, which leveraged a database to store information about the open-source projects (called repositories metadata information). In the first version, the database was a SQL DB which stored data in a dedicated table called *data fetcher* in a Json format based on a DOAP model [12]. In the second version, the storage functionality is provided by the KnowledgeBase, which finally provides DOAP data to the other services of the MORPHEMIC platform, such as the Profiler.

Table 1 The changes of the KnowledgeBase

Changed	Unchanged
Data Indexing	DOAP Model
Data Searching	Each document is provided in a JSON format

From a practical point of view, the Crawler has changed only with regards to the use of the tool needed to communicate with the KnowledgeBase (this process will be described in the next section: KnowledgeBase).
Other updates in the last Crawler prototype are related to the implementation of the repositories analysed and reported in the in the first period of the MORPHEMIC project, as Eclipse, C-Tan, R-Forge, J-Query plugin (added to the git-Hub and Apache Crawler repositories) [5]. Figure 6 shows the new process of the Crawler and how it works:

Figure 6 How to work the Crawler

The first step of the Crawler is to search and download the project information from specific repositories configured in the Crawler (Crawler repositories) as GitHub, j-Query, Apache, R-forge and so on [5].
These metadata information, in the last release of the Crawler aren't stored in a dedicated Data Warehouse[3] but are captured and sent in real time to the KnowledgeBase Storage in a Json data format. Specifically, every time that the Crawler downloads the projects metadata information (called data repositories in MORPHEMIC platform) from the Crawler repositories the KnowledgeBase creates/updates its index in real-time. The projects metadata information retrieved from the configured repositories will now be called repositories metadata information in MORPHEMIC project. The communication protocol used by the Crawler and the KnowledgeBase is a REST APIs that leverages the HTTP interface of Elasticsearch.

A test has been conducted to verify how the separation of the KnowledgeBase from the Crawler improved the times required to query the huge amount of data of the open-source projects scraped and processed.

Figure 7 Time Query Data

This benchmark (Figure 7) has been executed on 63327 repositories metadata information projects, filtered for topics of the last 3 months. The result shows that a clear improvement from the point of view of the scalability and capacity of the processed data has been achieved: the data are stored in real-time; at any time, the data can be used from the Profiler, the data can be used in simple and complex context (one or more processors, memories or devices).
The Crawler doesn't communicate directly with the Profiler, but it interacts directly with the KnowledgeBase.The interaction from Crawler and KnowledgeBase provided in the Polymorphic Adaptation processes will be described in the "D3.4 - Planning and adaptation results" [10].

KnowledgeBase

The KnowledgeBase can be considered as the second step of the classification approach to define the profiling application for the best deployment application model.The first prototype of the KnowledgeBase is provided as an internal component of the Crawler component [3].

The last prototype of the KnowledgeBase is based on Elasticsearch [14] as a distributed document store to optimize, update and search the repositories metadata information crawled. Elasticsearch is an open-source search engine based on the Lucene search server, with Full Text search capabilities and support for a distributed architecture. All its functionalities are natively exposed through a RESTful interface, while the information is stored in Json[4] data format. The implementation and integration of the KnowledgeBase in the MORPHEMIC platform helps taking advantage of the wealth of open-source applications in order to improve its Polymorphic Adaptation capabilities as well as the scalability of the platform where the multiple servers exploit a huge amount of data. Specifically, the main functionality of the KnowledgeBase is:

- To store every kind of data (textual, alpha-numeric, geospatial), structured and non-structured, in a NoSQL collection serialized as Json document.
    - the model used to retrieve and store the data sent (from the Crawler) is based on DOAP model provided in the first part of the MORPHEMIC project[3].
- To filter only the metadata information as requested from the other components of the polymorphic task, such as the Profiler.
- To send data to the Profiler through the REST API protocol of Elasticsearch.

The data schema (in a Json format) for the repositories metadata information project of the KnowledgeBase is based on the DOAP model [3] and indexed in an unstructured data in Json format as shown in the following figure:

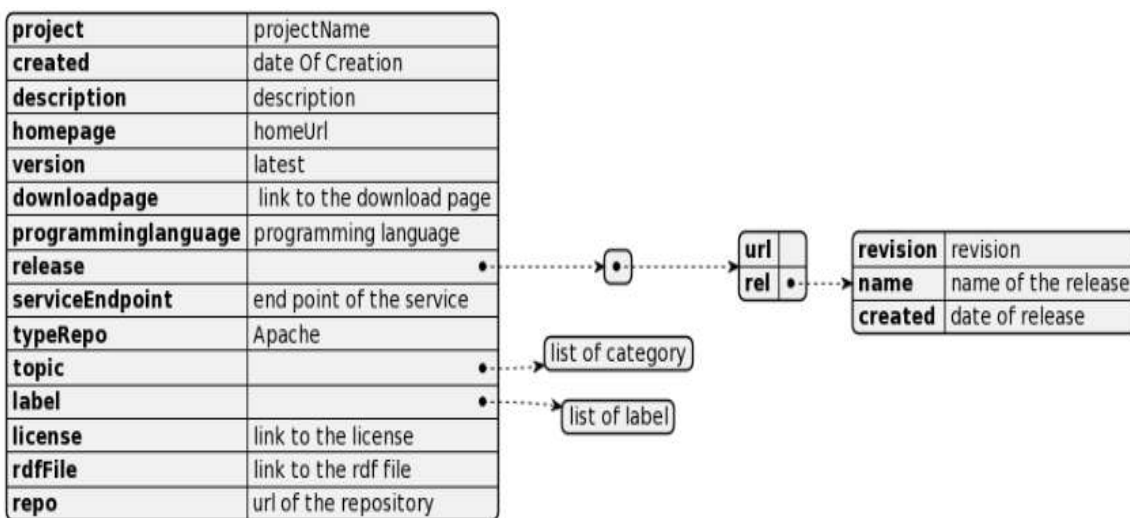

Figure 8 KnowledgeBase data schema

The following sequence diagram (Figure 9) shows the KnowledgeBase process: the KnowledgeBase receives the repository metadata information from the Crawler. The second step is to filter them from the Bag of Words (FPGA, server, Docker, GPU, Kubernetes, serverless). At this point, the KnowledgeBase stores this metadata information project received (from the Crawler in a JSON data format) into the NoSQL Database of the Elasticsearch: now it is ready to be sent to the Profiler. Specifically, at the end of this process the KnowledgeBase listens for the Profiler requests using the REST APIs of Elasticsearch.

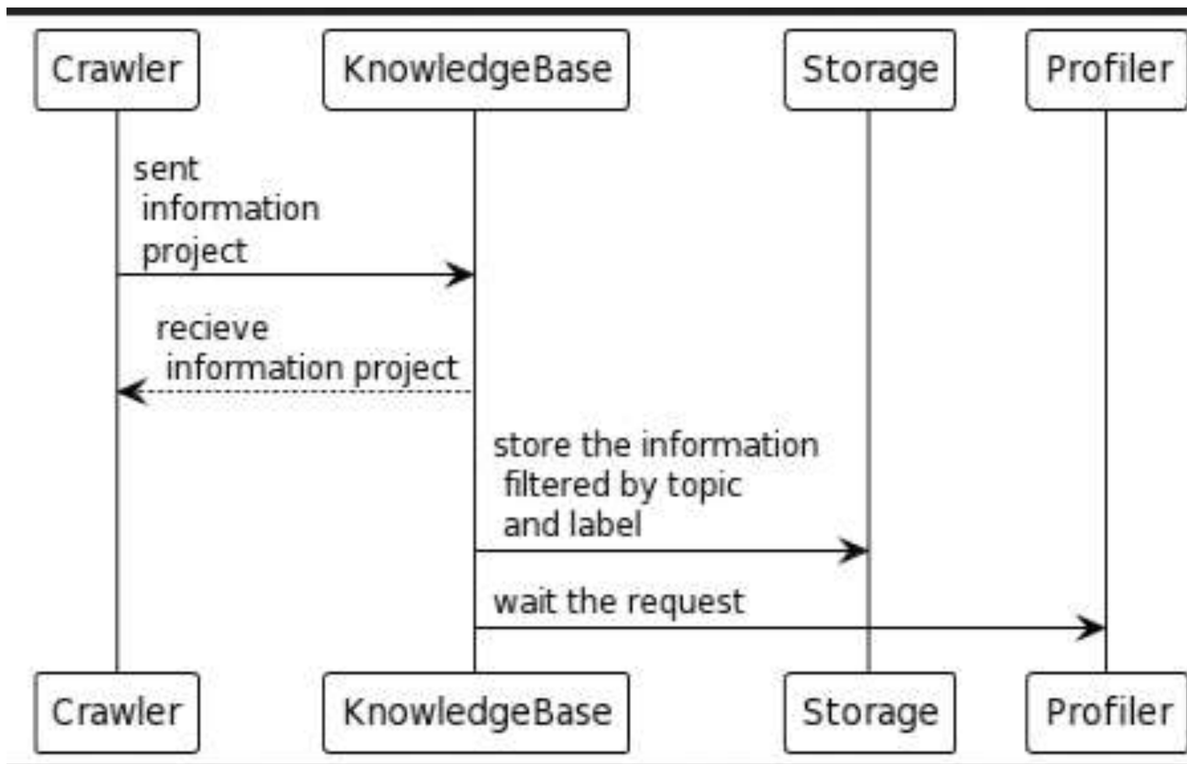

Figure 9 The sequence diagram of the KnowledgeBase

An example how the KnowledgeBase sends the data to the profiler is provided in the following example:

POST http://knowledgebase:9200/knowbase/_search/?q=java returns a list of repositories data as follows:


```
{
"_index": "knowbase",
"_type": "_doc",
"_id":"20",
"_score":1.0,"
"_source": {
"project": "Apache OpenNLP",
"created":"",
"description": "Apache OpenNLP software supports the most common NLP tasks",
"homepage": "http://opennlp.apache.org",
"version":"2.1.0",
```

```
"downloadpage": "http://opennlp.apache.org/download.html",
"programminglanguage":"Java",
"release": {
"url":"",
"rel": {
"revision":"2.1.0",
"name":"Apache OpenNLP",
"created":"2022-11-23"}},
"serviceEndpoint":"",
"typeRepo":"Apache",
"topic": ["library"],
"label":"opennlp",
"license": {"url": "http://usefulinc.com/doap/licenses/asl20"},
"rdfFile": "http://opennlp.apache.org/doap_opennlp.rdf",
"repo": ["https://github.com/apache/opennlp.git","http://svn.apache.org/repos/asf/opennlp"]}
}
```


The information provided regarding the index that contains the data repositories (in this case *knowbase*), the position of the data (*id*) and the data repositories information as name of the project, date of creation, last release, programming language (collected in the field *_source*). The indeed repositories metadata information is searchable in real-time. Specifically, the KnowledgeBase uses a data structured called index (knowledgebase) where every type of data is stored as a document. Each document indexed is a collection of fields (name of project, topic, label, description, release and so on). This type of index allows the processing of the repositories metadata in real-time, making the KnowledgeBase scalable and fast optimised for the acquiring, enriching, archiving, analysing, visualising of massive amount of data repositories (from the Crawler), and capable of sending this data to the Profiler, in real-time. The Profiler (downloader component) sends its requests by GET method (the source code of the Crawler, Profiler is available on the MORPHEMIC repository [15]):

```
...........
response = requests.get(url=url_knowledge_base+'/knowbase/_search/', headers=headers)
.............
```

The sequence events that explain how the KnowledgeBase communicates with the Profiler will be explained in the Polymorphic Adaptation processes in the MORPHEMIC deliverable “D3.4 - Planning and adaptation results” [10].

3. Code classification results: Classifier

3.1 Introduction

Code Classification allows 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 categorized 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.

3.2 Results about the feature definition process

The section provides the feature definition process and how this process is defined according to the characteristics of the MORPHEMIC code mining. The feature selection is required for code classification. The code classification helps to define an application profile, being used to obtain the best possible adaptation of the application deployment to suitable infrastructures and component configurations. The first step was to identify and analyse the tools for the code classification; the results of this study and analysis is provided in the MORPHEMIC deliverable “D3.1-Software tools and repositories for the code mining”[3]: 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). The focus of the feature extraction, in this section, is on analysing the code graph of the application. The analysis results in a set of properties from which we select the features relevant for the training process in classification. Figure 10 shows the different forms of such a graph.

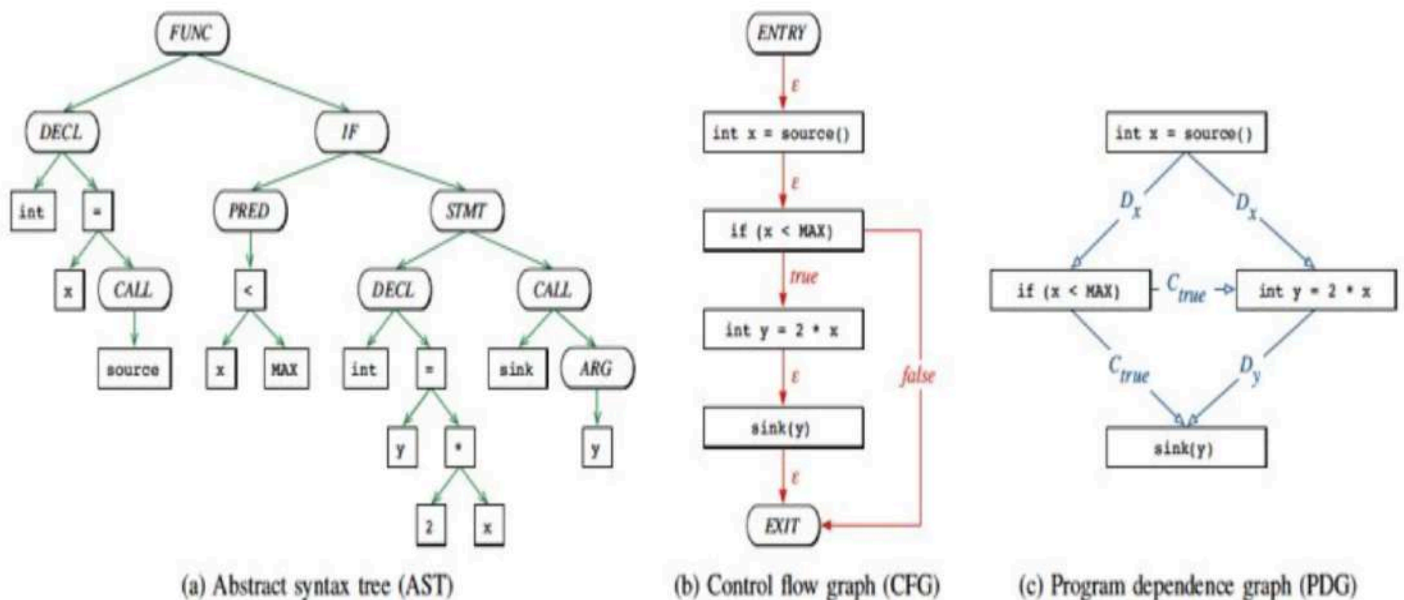

Figure 10 Different forms of representing code using graphs

As shown in Figure 11 , by constructing the concrete syntax tree (CST) or parse tree (bottom left), which is retranslated into an abstract syntax tree (AST) (upper left). To analyse the program's Control flow, we generate a control flow graph (CFG) from the abstract syntax tree. Based on the information it contains, we can determine control- and data dependencies as expressed by the dominator tree (DT), the post-dominator tree (PDT), and finally, the program dependence graph (PDG), which is constructed by combining information from the control flow graph and the post dominator tree.

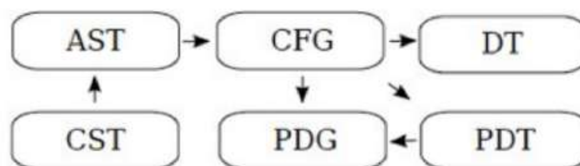

Figure 11 Dependencies between program representations

The Code Property Graph (CPG) merges graphs from the compiler such as the Abstract Syntax Tree, the Program Dependence Graph, the Control Flow graph, etc., into a single joint data structure. The Code Property Graph is a data structure designed to mime large codebases for instances of programming patterns. These patterns are formulated in a domain-specific language (DSL) based on Scala [16]. It serves as a single intermediate program representation across

all languages supported by Ocular. The property graph is stored in a graph database, as for example ArangoDB [17], Neo4J [18] and OrientDB[19], and made accessible via a domain specific language (DSL) for the identification of programming patterns - based on a DSL for graph traversals. Then we can get the features at three levels: link-level, node-level and graph-level, as shown in the following figure:

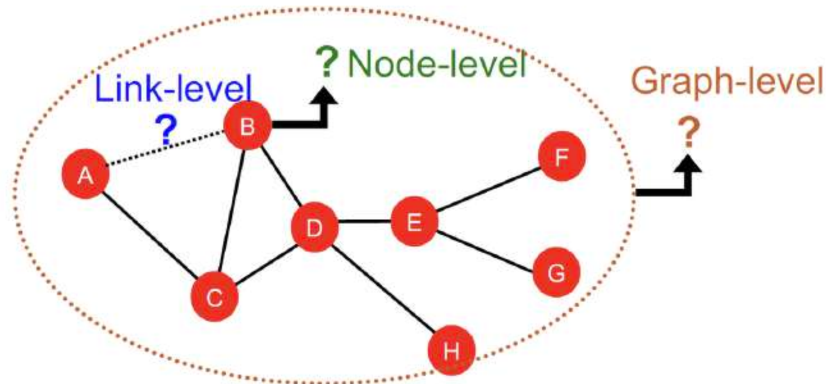

Figure 12 Feature extraction from the code property graph

3.2.1 Evaluation of the code classification tools

Based on this study, the first step has been to identify a corpus of software project code representative for code classes: the High-Performance Computing (HPC) code or web code. Another step was to identify the tools to use for this classification. It decided to use the graph tools because 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.

Table 2 Evaluation of the code classification tools

Tool	State	Support Language	Visualisation	Link
ConQAT	Not active after 2018 as commercialize, but academic free	Many	Dashboard: metric values, trends, tables, charts, and tree maps.	https://www.cqse.eu/en/teamscale/features/
Moose	active	Mostly objective orient like Java; You can build your own engine	Graph, need Pharo launcher	https://github.com/moosetechnology/Moose
Code Climate	Also commercialize, but some open source	Many		https://codeclimate.com/

SourceMeter	commercialize, but some plugin open source	C/C++, Java, C#, Python, and RPG projects ...		https://github.com/FrontEndART/SonarQube-plug-in

We represent the program code by the tool, Joern[20], as a format of Code Property Graph, which is a combination of Abstract Syntax Tree, the Program Dependence Graph, the Control Flow graph, etc., into a single joint data structure.

The Code Property Graph is a data structure designed to mime large codebases for instances of programming patterns.

These patterns are formulated in a domain-specific language (DSL) based on Scala [16]. It serves as a single intermediate program representation across all languages supported by Ocular. The property graph is stored in a graph database, such as ArangoDB [17], Neo4J [18] and OrientDB[19], and made accessible via a domain-specific language (DSL) to identify programming patterns - based on a DSL for graph traversals. We encode graph-based structures in an appropriate format that is amenable to machine learning. We will test and evaluate different machine learning techniques to find the optimal one, such as deep learning, reinforcement learning, graph neural network (GNN), etc. As a result, a graph-based classification model that is ultrafast can be deduced to support the accurate, graph-based classification of open-source software components.

3.3 Classifier

3.3.1 Internal Architecture

The internal architecture of the Classifier can be seen in Figure 13. It comprises three categories of components:

1. classification components (BoWClassifier, GraphClassifier, Compositor).
2. a component deciding when to classify and when to update the classification model (Model Updater).
3. a component that exposes the main functionalities of the Classifier to ease its integration with the MORPHEMIC platform (REST Controller).

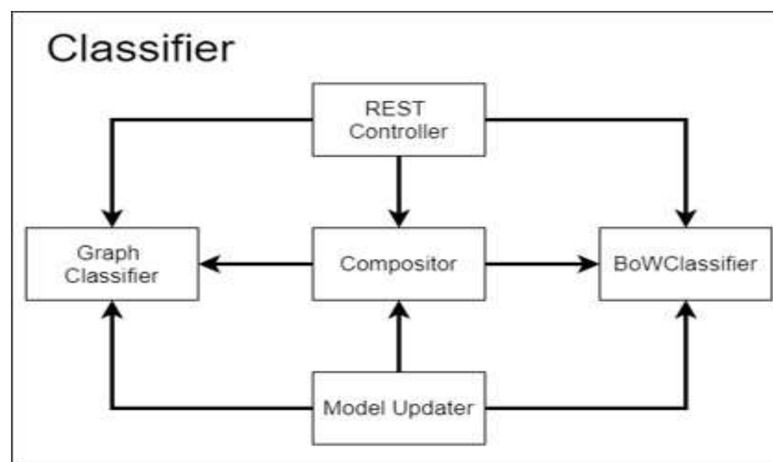

Figure 13 Code Classifier design

3.3.2 Functional Analysis

The REST Controller exposes parts of the main functionality that can be delivered by the Classifier. It enables applying the current classification model of a specific classification algorithm that is encapsulated by the respective classification-based component (Graph Classifier, BoWClassifier, or Compositor). In addition, it could enable to execute re-classifications, i.e., updating of the classification models upon a respective request.

The Model Updater is a key component in the architecture as it needs to take critical decisions about when to start the classification and when to update the classification models. To take such decisions, it relies on the Knowledge Base, in which the extracted features of open-source repositories will be stored as well as updates on these features (all done by the Analyser). As such, the Knowledge Base can be regarded as an indirect communication medium between the Classifier and the Analyser.

The decisions taken are forwarded to the classification-based components that need to create a new or update their existing classification model, based on the respective algorithm or technique that they apply. As the Model Updater is the component responsible for the communication with the Knowledge Base, it also takes care of classifying the open-source repositories based on the classification models derived/updated. This then enables the matching of (application) components to those open-source components that have been crawled.

3.3.3 Technical Analysis

The Classifier follows a multi-threaded architecture encompassing libraries offering threads that provide the required classification functionality. The threads will be obviously coordinated by the Model Updater. Due to the use of the multi-threaded architecture, it is proposed that the implementation relies on a programming language that provides suitable, complete support for multi-threaded programming which in this case is Python [15]. The whole Classifier component will be offered in the form of a container image and will be thus executed as a micro-service.

3.4 Evaluation of the Code Classification Tool

The Code Classification is presented by Graph Neural Network (CCGNN): a general-purpose processing pipeline geared towards classifying the application codes in a robust and learnable manner. The pipeline, depicted in Figure 14, accepts code in various source languages and converts them to LLVM (Low Level Virtual Machine) Intermediate Representation (IR)[21]. We explain the details of LLVM IR in Section III-B. Then we utilize Joern [20] and llvm2cpg to process the IR code and convert them to CPG. We explain the details of Joern documentation as provided in the deliverable "D3.4 Planning and adaptation results" [10]. CPG is composed of abstract syntax, data flow and control flow of code, so it naturally supports loops and function calls. In turn, the CPG structure is used to train the embedding space of a single sentence, called inst2vec (from the word" instruction"), which is fed to the RNN [23] to perform code classification tasks. The basic workflow is the following:

- Convert any programs into LLVM IR.
- Generate a CPG using llvm2cpg and Joern.
- Train the embedding space of a single sentence, called inst2vec.
- Start the code classification using RNN.

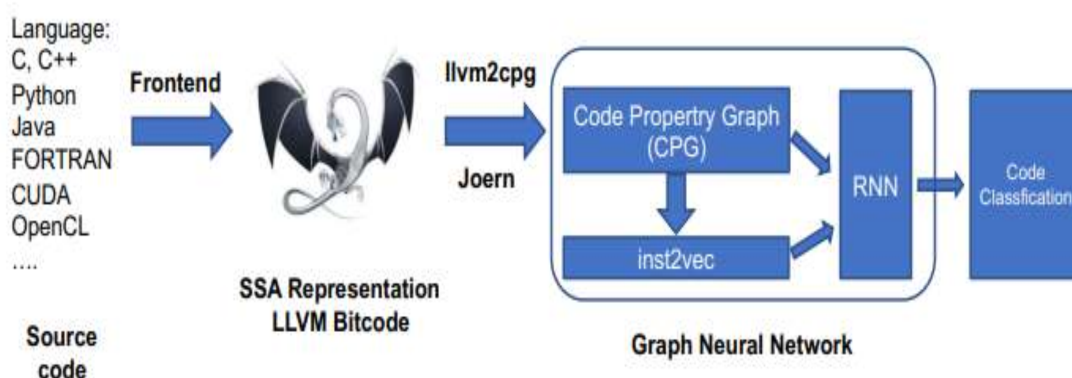

Figure 14 The pipeline for code classification by graph neural network

3.4.1 LLVM

The LLVM project was initially developed in 2000 by Vikram and Chris of UIUC (University of Illinois Urbana-Champaign). Their original purpose was to develop a dynamic compilation technique for static and dynamic programming languages. In 2005, Chris joined Apple and continued the development of LLVM. In 2013, Sony also started using CLang to develop PS4. LLVM originally stood for Low-Level Virtual Machine, but as the LLVM family grows larger, this original meaning is no longer applicable. The LLVM project contains modular, reusable compilers and toolchains. Eleven LLVM sub-projects are listed on the official website. Here are the two most important ones:

- LLVM core optimizer (optimizer). It does not depend on the source code and target machine instruction set. This core library is based on the LLVM intermediate representation LLVM (IR).
- Clang compiler (compiler). The goal of Clang is to implement a C/C++/Objective-C compiler. Clang can provide very useful error and warning messages. Clang Static Analyzer is a tool to automate bugs. Before introducing LLVM IR, we need to understand the structure of LLVM. The traditional static compiler is divided into three stages: front-end, optimization, and back-end, as shown in the following figure:

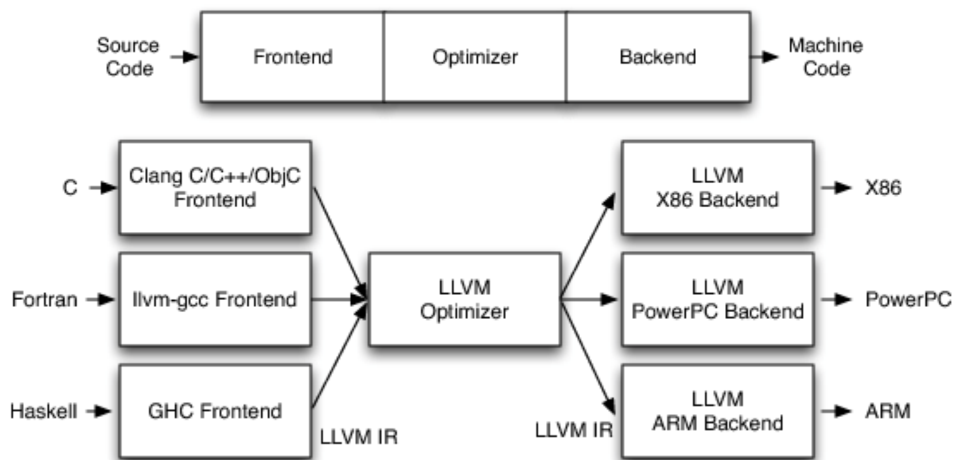

Figure 15 Three phase design three-phase compiler

The Three-Phase Compiler is a traditional compiler design. It is widely used, for example, JIT compilers or interpreters use this structure. Java Virtual Machine also uses this structure. The biggest advantage of this design is the coupling, allowing each part to focus on its own function. For example, if all languages use the same intermediate representation, it means that the same Optimizer can be used. This greatly improves code reuse and Optimizer optimization. The three-stage design of LLVM is like this: The advantage of this is that if we need to support a new programming language, then we only need to implement a new front end. If we need to support a new hardware device, then we only need to implement a new backend. The optimization stage is for the unified LLVM IR, so it is a general stage. There is no need to modify the optimization stage, whether it is to support a new programming language or a new hardware device. Thus, the role of LLVM IR is clear here. LLVM IR has three main formats: one is a compiled intermediate language in memory; one is a binary intermediate language stored on the hard disk (ends with .bc), and the last is a readable intermediate format (ends with .ll) . These three intermediate formats are completely equal. LLVM IR is the key to LLVM optimization and code generation. According to the readable IR, we can know what kind of code we have generated before finally generating the target code. And according to IR, we can choose to use different backends to generate different executable codes. At the same time, because a unified IR is used, we can reuse the optimization features of LLVM, even if we use a programming language designed by ourselves.

3.4.2 Joern

Joern is used in academic research as a source for intermediate graph representations of code, particularly in machine learning and vulnerability discovery applications [20]. To support this use-case, Joern provides both plotting capabilities in the interactive console as well as the joern-export command line utility. We represent the program code by the tool, Joern, as a format of Code Property Graph (CPG). CPG merges graphs from the compiler such as the Abstract Syntax

Tree, the Program Dependence Graph, the Control Flow graph, etc., into Figure 16. Exemplary code sample for Joern. a single joint data structure.

```
void foo()                  1
{                           2
    int x = source();       3
    if (x < MAX)            4
    {                       5
        int y = 2 * x;      6
        sink(y);            7
    }                       8
}                           9
```

Figure 16 Code example for Joern

The Code Property Graph is a data structure designed to mime large codebases for instances of programming patterns.

These patterns are formulated in a domain-specific language (DSL) based on Scala [16]. It serves as a single intermediate program representation across all languages supported by Ocular. The property graph is stored in a graph database, as for example ArangoDB [17], Neo4J [18] and OrientDB [19] , and made accessible via a domain specific language (DSL) for the identification of programming patterns - based on a DSL for graph traversals. We focus on three classic representations, namely abstract syntax trees, control flow graphs and program dependence graphs which form the basis for our approach for code classification. As a simple example illustrating the different representations and running through this section, we consider the code sample shown in Figure 10.

Abstract syntax trees are usually among the first intermediate representations produced by code parsers of compilers and thus form the basis for the generation of many other code representations. These trees faithfully encode how statements and expressions are nested to produce programs. Abstract syntax trees are ordered trees where inner nodes represent operators (e.g., additions or assignments) and leaf nodes correspond to operands (e.g., constants or identifiers). As an example, consider Figure 10a showing an abstract syntax tree for the code sample given in Figure 16 a control flow graph explicitly describes the order in which code statements are executed as well as conditions that need to be met for a particular path of execution to be taken. To this end, statements and predicates are represented by nodes, which are connected by directed edges to indicate the transfer of control. While these edges need not be ordered as in the case of the abstract syntax trees, it is necessary to assign a label of true or false to each edge. A statement node has one outgoing edge labelled as, whereas a predicate node has two outgoing edges corresponding to a true or false evaluation of the predicate. Control flow graphs can be constructed from abstract syntax trees in a two-step procedure: first, structured control statements (e.g., if, while, for) are considered to build a preliminary control flow graph. Second, the preliminary control flow graph is corrected by additionally considering unstructured control statements such as goto, break and continue. Figure 10b shows the CFG for the code sample given in Figure 17. Program dependence graphs introduced by Ferrante et al. [22] have been developed initially to perform program slicing, that is, to determine all statements and predicates of a program that affects the value of a variable at a specified statement. The program dependence graph explicitly represents dependencies among statements and predicates. In particular, the graph is constructed using two types of edges: data dependency edges reflecting the influence of one variable on another and control dependency edges corresponding to the influence of predicates on the values of variables. The edges of a program dependence graph can be calculated from a control flow graph by first determining the set of variables defined and the set of variables used by each statement and calculating reaching definitions for each statement and predicate, a standard problem from compiler design. As an example, Figure 10c shows the program dependence graph for the code sample given in Figure 16.

3.5 Build the hierarchical graph

As an example, Figure 17 shows when building a graph, it's often useful to have the ability to embed nodes within other nodes and to cluster nodes together hierarchically. This allows us to represent more complex structures and relationships within our data.

Embedding nodes within other nodes is sometimes called "nesting" or "grouping". This allows us to group related nodes together and represent them as a single entity. Hierarchical clustering is a way of grouping nodes together based on their similarity or distance from each other. This allows us to represent the data in a way that reveals underlying patterns and relationships. Together, embedding nodes and hierarchical clustering allow us to build more complex and informative graphs. They allow us to represent relationships and patterns within our data that might be difficult to see with a simple graph. By using these techniques, we can gain deeper insights into our data and make more informed decisions.

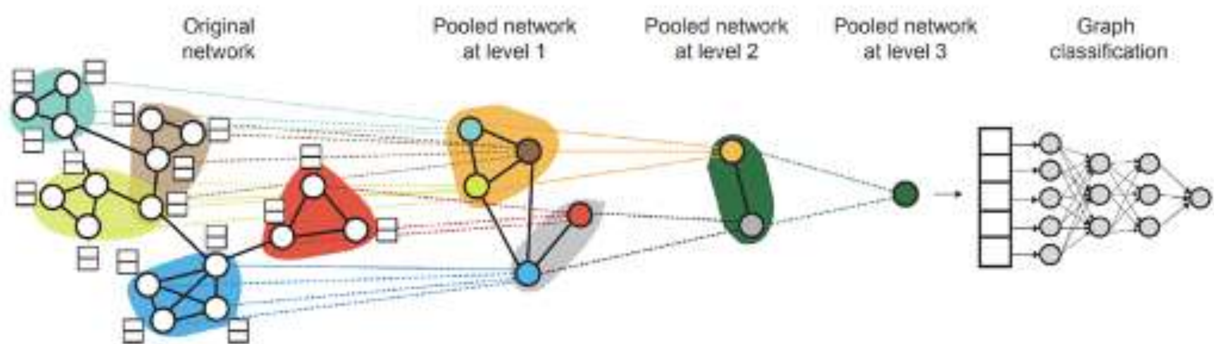

Figure 17 Embed nodes within other nodes and to cluster nodes together hierarchically

3.5.1 Inst2vec

Inst2vec [23] is a technique for training the embedding space of a single sentence. The goal of inst2vec is to represent a sentence as a high-dimensional vector that captures its semantic meaning, allowing us to compare and analyse sentences in a more meaningful way. To train the inst2vec model, we start by representing each word in the sentence as a vector. These word vectors are initialized randomly and then updated during training to maximize the likelihood of the sentence given the context of other sentences in the corpus. This is typically done using a neural network architecture, such as a recurrent neural network (RNN) or a transformer.

Once the word vectors have been learned, we can use them to compute a vector representation of the entire sentence.

One common approach is to simply average the word vectors, although more sophisticated techniques can also be used, such as hierarchical pooling or attention mechanisms. The resulting sentence vector can be used for a variety of tasks, such as sentiment analysis, classification, or similarity comparisons. For example, we could compare the similarity between two sentences by computing the cosine similarity between their corresponding sentence vectors.

One advantage of inst2vec is that it allows us to train a sentence embedding space in an unsupervised manner, meaning we don't need labelled data to train the model. This can be particularly useful in situations where labelled data is scarce or expensive to obtain.

3.5.2 Combining Inst2vec and RNN

Combining inst2vec and RNN can be a powerful approach for code classification, particularly for distinguishing between different types of code such as CPU, GPU, and FPGA code.

To use this approach, we would start by training an inst2vec model to learn high-dimensional embeddings for each code snippet. Each code snippet could be represented as a sequence of tokens, such as keywords, variable names, and function names. Once we have learned the embeddings for each code snippet, we could use them as input to an RNN classifier.

The RNN could be trained to classify each code snippet into one of several categories, such as CPU, GPU, or FPGA code. During training, we would feed each code snippet into the inst2vec model to obtain its embedding, and then feed the sequence of embeddings into the RNN. The RNN would then output a probability distribution over the different categories. At test time, we would use the trained inst2vec and RNN models to classify new code snippets. We would first obtain the embeddings for the new code snippet using the inst2vec model, and then feed the sequence of embeddings into the RNN to obtain the final classification.

3.6 Results

One advantage of this approach is that it allows us to capture both the syntax and semantics of the code snippet, which can be important for distinguishing between different types of code. Additionally, because the inst2vec model is trained in an unsupervised manner, we can leverage large amounts of unlabelled code to learn the embeddings, which can be particularly useful in domains where labelled data are scarce.

Next, we present how we use inst2vec and RNN to classify between several types of code such as CPU, GPU, and FPGA code:

- **CPU code classification**: Given a code snippet that performs some computation using the CPU, we could use inst2vec and RNN to classify it as CPU code. The inst2vec model would learn embeddings for each token in the code snippet, and the RNN would be trained to distinguish between CPU code and other types of code. An example of a CPU code snippet could be a function that implements matrix multiplication using loops and scalar arithmetic.
- **GPU code classification**: Similarly, we could use inst2vec and RNN to classify code snippets that are written to be executed on a GPU. An example of a GPU code snippet could be a function that performs matrix multiplication using CUDA kernels and GPU memory operations.
- **FPGA code classification**: FPGA code is typically written in hardware description languages such as Verilog or VHDL. We could use inst2vec and RNN to classify code snippets that are written in these languages as FPGA code. An example of an FPGA code snippet could be a Verilog module that implements a custom hardware accelerator for matrix multiplication.

Figure 18 is an example of how the inst2vec and RNN approach could be used to classify between different types of code, using a dataset of labelled code snippets:

Code Snippet	Label
int main() { return 0; }	CPU
global void matrixMult(float *A, float *B, float *C, int width) { ... }	GPU
module matrix_mult(input clk, input reset, input [15:0]A [15:0]B, output [15:0]C) { ... }	FPGA
for (int i=0; i<n; i++) { for (int j=0; j<n; j++) { C[i][j] = 0; for (int k=0; k<n; k++) { C[i][j] += A[i][k] * B[k][j]; } } }	CPU
global void convolution(float *input, float *output, int input_width, int filter_width) { ... }	GPU
module adder(input a, input b, output c) { ... }	FPGA

Figure 18 Data label code snippets

In this example, we have a dataset of six labelled code snippets, two each for CPU, GPU, and FPGA code. The inst2vec model and RNN classifier would be trained on this dataset, and then used to classify new code snippets into one of these

three categories. The exact architecture of the inst2vec model and RNN classifier are implemented using deep learning frameworks of PyTorch.

Code classification model using PyTorch:

Here's an example of how to implement the code classification model using PyTorch:

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define the Inst2vec model
class Inst2vec(nn.Module):
   def __init__ (self, vocab_size, embedding_dim, hidden_dim):
      super (Inst2vec, self). __init__ ()
      self.embedding = nn.Embedding(vocab_size, embedding_dim)
      self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)

   def forward (self, x):
      x = self.embedding(x)
      _, (h_n, _) = self.lstm(x)
      return h_n[-1]

# Define the RNN classifier
class Classifier(nn.Module):
   def __init__ (self, input_dim, hidden_dim, output_dim):
      super (Classifier, self). __init__ ()
      self.fc1 = nn.Linear(input_dim, hidden_dim)
      self.fc2 = nn.Linear(hidden_dim, output_dim)
      self.softmax = nn.LogSoftmax(dim=1)

   def forward (self, x):
      x = self.fc1(x)
      x = torch.relu(x)
      x = self.fc2(x)
      x = self.softmax(x)
      return x

# Define the complete model
class CodeClassifier(nn.Module):
   def __init__ (self, vocab_size, embedding_dim, hidden_dim, output_dim):
      super(CodeClassifier, self). __init__ ()
      self. inst2vec = Inst2vec(vocab_size, embedding_dim, hidden_dim)
      self.classifier = Classifier(hidden_dim, hidden_dim, output_dim)

   def forward (self, x):
      x = self. inst2vec(x)
      x = self.classifier(x)
      return x

# Example usage:
vocab_size = 10000 # Size of the vocabulary
embedding_dim = 100 # Dimension of the token embeddings
hidden_dim = 128 # Hidden size of the LSTM
output_dim = 3 # Number of output classes
model = CodeClassifier(vocab_size, embedding_dim, hidden_dim, output_dim)
criterion = nn.NLLLoss() # Negative Log Likelihood loss for classification
optimizer = optim.Adam(model.parameters(), lr=0.001) # Adam optimizer for training
```

```
# Example training loop:
for epoch in range(num_epochs):
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion (outputs, labels)
        loss.backward()
        optimizer.step()
    print ('Epoch {}, Loss: {}'. format (epoch+1, loss.item()))

# Example prediction:
inputs = torch.tensor([[1, 2, 3, 4, 5]]) # Example input code snippet
with torch.no_grad():
    outputs = model(inputs)
    _, predicted = torch.max(outputs.data, 1)
print ('Predicted label:', predicted.item())
```

In this example, we define the Inst2vec model as an LSTM that takes as input a sequence of token IDs and outputs the hidden state of the final time step. We then define the RNN classifier as a two-layer fully connected network with ReLU activation and a "*LogSoftmax*" output (as provided in the code above). We combine these two models to create the Code Classifier model, which takes as input a sequence of token IDs and produces a classification output.
We also define the training loop using an Adam optimizer and a negative log likelihood loss function and perform an example prediction on an input code snippet. Note that the specific architecture and hyperparameters will need to be tuned for the specific task and dataset.

3.7 Classification results in BoW

The classification mechanism consists of two steps:

- the clusters creation based on labels (L') and
- the prediction of cluster given the programming and the feature list.

Cluster creation

After the feature preparation and relabeling process, we have started by applying a PCA (Principal component analysis) technique consisting of selecting the n best feature representing a document. Since repositories don't have necessarily the same number of tokens, we've used PCA not only for reducing the feature size but also for uniformizing the size of all features list.. The vector TFIDF is then created representing the input or X vector. The Y vector or the result is the expected result, as illustration a repository detected as GPU and Docker will produce the output vector: 1 0 1 0 when a repository classified as FPGA and Docker will correspond to the vector 0 1 1 0. The X and Y vectors will be fed to Sequential neural network for training. The latter will be stored for the prediction phase.

Prediction

The prediction phase reproduces the compression and TFIDF process of the training phase with the same parameters. The X vector created will be then send to the model for prediction.

We have experimented with 25,000 repositories divided by program language. For every collection (repositories divided by programming language), train and test samples will be extracted for measuring the accuracy. By accuracy we mean, the precision in predicting the repository label (variant or hardware). The following table shows the results:

Table 3 Test results

Language	Accuracy	Rmse	r2score	size
Python	0.989	0.035	0.982	7551
Java	0.985	0.039	0.970	8038
JS	0.982	0.044	0.964	7898

Cpp	0.978	0.046	0.874	7444
Go	0.944	0.084	0.912	7685

3.8 Evaluation metric for classification models

The most common evaluation metric for classification models is accuracy, which measures the proportion of correct predictions over the total number of predictions. To compute accuracy, we can use the following code after training the model:

```
correct = 0
total = 0
with torch.no_grad ():
  for inputs, labels in test_loader:
    outputs = model(inputs)
    _, predicted = torch.max (outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels). sum (). item ()

print ('Test Accuracy: {} %'. format (100 * correct / total))
```

This code iterates through the test data and computes the predicted labels using the trained model. It then compares the predicted labels with the ground truth labels to compute the number of correct predictions. Finally, it computes the accuracy by dividing the number of correct predictions by the total number of predictions.

Additionally, we can also compute other evaluation metrics such as precision, recall, and F1 score for each class using the scikit-learn library in Python. For example:

```
from sklearn.metrics import classification_report

with torch.no_grad():
  y_true = []
  y_pred = []
  for inputs, labels in test_loader:
    outputs = model(inputs)
    _, predicted = torch.max(outputs.data, 1)
    y_true.extend(labels.numpy())
    y_pred.extend(predicted.numpy())

print(classification_report(y_true, y_pred))
```

This code computes the precision, recall, F1 score, and support (number of samples) for each class using the ground truth labels and the predicted labels. These metrics provide a more detailed view of the performance of the model for each class.

Suppose we have a dataset with three classes: CPU, GPU, and FPGA codes, and we trained a code classification model using the inst2Vec and RNN architecture. We evaluated the model using a test set of 1000 code snippets, with 300 CPU, 400 GPU, and 300 FPGA codes. Here is an example results table:

Table 4 Evaluation results

Class	Precision	Recall	F1 Score	Support
CPU	0.91	0.88	0.89	300
GPU	0.95	0.97	0.96	400
FPGA	0.92	0.91	0.91	300
Total	**0.93**	**0.93**	**0.93**	**1000**

In Table 4, we can see the precision, recall, F1 score, and support for each class as well as the overall metrics for the entire dataset. We can see that the model achieved high performance for all three classes and an overall accuracy of 93%.

- **Class**: This column lists the different classes in the classification task. In this example, we have three classes: CPU, GPU, and FPGA codes.
- **Precision**: Precision measures the proportion of correctly predicted instances of a given class over all instances that were predicted as that class. In other words, it tells us how often the model was correct when it predicted a specific class. In this example, we can see that the model achieved a precision of 0.91 for CPU codes, 0.95 for GPU codes, and 0.92 for FPGA codes.
- **Recall**: Recall measures the proportion of correctly predicted instances of a given class over all instances that belong to that class. In other words, it tells us how often the model correctly identified a specific class out of all the instances of that class. In this example, we can see that the model achieved a recall of 0.88 for CPU codes, 0.97 for GPU codes, and 0.91 for FPGA codes.
- **F1 Score**: The F1 score is the harmonic mean of precision and recall. It provides a single score that balances both precision and recall and is often used as a summary metric for classification performance. In this example, we can see that the model achieved an F1 score of 0.89 for CPU codes, 0.96 for GPU codes, and 0.91 for FPGA codes.
- **Support:** The support column lists the number of samples in each class in the test set. In this example, we have 300 CPU codes, 400 GPU codes, and 300 FPGA codes in the test set.
- **Total**: The total row provides the overall evaluation metrics for the entire test set. In this example, we can see that the model achieved an overall precision of 0.93, an overall recall of 0.93, and an overall F1 score of 0.93 for the entire dataset.

4. Conclusion

This deliverable provides the final shape of the Application Profiling task, in the MORPHEMIC project, and its implementation: Profiler. Profiler works offline, before the application deployment, interoperates with CAMEL Designer to find the initial deployment model, and helps to the Camel Model; the code mining tools to search and find the metadata of open-source projects and the Classifier to find the best application deployment model based on the Graph techniques (e.g., LLVM, Joern).

Specifically, to obtain the best deployment adaptation of the application deployment to suitable infrastructures and component configurations, the definition of the Application Profiling is based on the code classification according to project metadata as provided from the code mining tools.

We adopt High-Performance Computing (HPC) as the best code classes for the definition of the Profiler. Specifically, we prove that the HPC can range from the qualitative feature like the programming languages used, structural features reflected in the application call graph, for example the way components and function invoke other components. Specifically, the Classifier is implemented using the code classification by Graph Neural Network (CCGNN): a general-purpose processing pipeline geared towards classifying the application codes in a robust and learnable manner. It results in combining inst2vec and RNN for distinguishing between different types of code such as CPU, GPU, and FPGA code. To do that, we start by training an inst2vec model to learn high-dimensional embeddings for each code snippet. Because the inst2vec model is trained in an unsupervised way, it allows us to leverage large amounts of unlabelled code to learn useful domains where labelled data are scarce. An important result of our approach is to be able to acquire both the syntax and semantics of the code snippet, which is important for distinguishing between different types of code.

Even though we achieved interesting and useful results out of the code classifier model presented in this deliverable (i.e., code graph-based), there are still other interesting directions for future work. One of them is obtaining an abstract model of the code graph which represents the application type as accurate as the actual code graph. In this way the training process can be further improved, and unnecessary nodes and vertexes that may reduce the precision can be removed. Moreover, applying transfer learning to the RNN-based classifier may improve the accuracy and reduce the process of learning, for example in the cases that two code categories have some key commonalities.

The static analysis of the code using the bag of words is an easy technique to implement, however, it requires the collection of a large quantity of data. The considerable number of data is an important requirement to be fulfilled however the quality of these data must be taken as another important parameter for better classification. We used GitHub APIs to collect repositories categorized under a certain label. These labels are chosen by the owner of the repository or by an algorithm with primitive technique. Therefore, this can lead to classification errors due to imprecision in the labeling of repositories. We used an approach consisting of combining unsupervised learning with supervised learning. We wish to continue to research for techniques that will further minimize the categorization error of repositories. We have created models by languages with almost 25,000 repositories. We strongly believe that our classifier will be better with a bigger number of repositories. Therefore, we plan to use more advanced techniques in training large models supporting a large amount of data. In fact, distrubuted techniques for training very large data that cannot fit in a single node.

5. References

[1] "MORPHEMIC Cloud," *morphemic cloud*. https://www.morphemic.cloud/

[2] Alessandro Rossini *et al.*, "The cloud application modelling and execution language (CAMEL)," *OPen Access Repos. Univ. Ulm*, p. 39, Mar. 2017, doi: 10.18725/OPARU-4339.

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

[4] Geir Horn and Paweł Skrzypek, "MELODIC: Utility Based Cross Cloud Deployment Optimisation," in *Proceedings of the 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, Conference Location: Krakow, Poland: IEEE Computer Society, May 2018, pp. 360–367. doi: 10.1109/WAINA.2018.00112.

[5] "Paragraph 4.3 - "Evaluation of the open-source project repositories of the deliverable" - D3.1-Software tools and repositories for the code mining," public.

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

[7] Yiannis Verginadis, Ioannis Patiniotakis, Christos Chalaris, Gregoris Mentzas, Kyriakos Kritikos, and Keith Jeffery, "D2.4 Metadata schema," Melodic Project Deliverable, Nov. 2017.

[8] Kaïs Chaabouni, Alexandros Raikos, Alicja Reniewicz, Andreas Tsagkaropoulos, Benjamin Leroy, Ferath Kherif, Maxime Compastié, Robert Gdowski, Alessandra Bagnato, Etienne Brosse, "D5.1 User Interfaces Specification," MORPHEMIC Project Deliverable, Aug. 2020.

[9] "Better Code Hub." https://bettercodehub.com/

[10] "D3.4 Planning and adaptation results."

[11] "Chapter 2 - Application Profiling - D3.1-Software tools and repositories for the code mining," public.

[12] "Chapter 3 - Code Mining Components - D3.1-Software tools and repositories for the code mining," public.

[13] Amina Moussaoui, Alessandra Bagnato, Etienne Brosse, Jean-DidierTotow, Adeliya Latypova, "D5.2 User Interface Guidelines," MORPHEMIC Project Deliverable, Aug. 2022.

[14] "Elasticsearch." https://www.elastic.co/pricing/faq/licensing

[15] "Profiler." [Online]. Available: https://gitlab.ow2.org/melodic/morphemic-preprocessor/-/tree/morphemic-rc4.0/profiler

[16] "Scala." http://scala-ide.org/docs/dev/index.html

[17] "Arango DB." https://www.arangodb.com/

[18] "Neo4j." https://neo4j.com/

[19] "OrientDB." http://orientdb.org/

[20] "Joern Documentation," 2021. https://docs.joern.io/home

[21] "LLVM Official WebSite," 2021. https://llvm.org/

[22] A. J. Ferrer *et al.*, "OPTIMIS: A holistic approach to cloud service provisioning," *Future Gener. Comp Syst*, vol. 28, no. 1, pp. 66–77, 2012, doi: 10.1016/j.future.2011.05.022.

[23] T. Hoefler, "T. Ben-Nun, A. S. Jakobovits, and Neural code compre- hension: A learnable representation of code semantics .," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, in NIPS'18.Red Hook, NY, USA:Curran Associates Inc. 2018.

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