H2020-ICT-2018-2-825040

Rational decomposition and orchestration for serverless computing

Deliverable 3.6

Defect prediction tool I

Version: 1.0

Publication Date: 30-June-2020

Deliverable Card

Deliverable	D3.6
Title:	Defect Prediction Tool I
Editor(s):	Stefano Dalla Palma (TJD)
Contributor(s):	Dario Di Nucci (TJD), Damian A. Tamburri (TJD)
Reviewers:	Vasilis Tountopoulos (ATC), Mike Long (PRQ)
Type:	Report
Version:	1.0
Date:	30-June-2020
Status:	Final
Dissemination level:	Public
Download page:	http://radon-h2020.eu/public-deliverables
Copyright:	RADON consortium

The RADON project partners

IMP	IMPERIAL COLLEGE OF SCIENCE TECHNOLOGY AND MEDICINE
TJD	STICHTING KATHOLIEKE UNIVERSITEIT BRABANT
UTR	TARTU ULIKOOL
XLB	XLAB RAZVOJ PROGRAMSKE OPREME IN SVETOVANJE DOO
ATC	ATHENS TECHNOLOGY CENTER ANONYMI BIOMICHANIKI EMPORIKI KAI TECHNIKI ETAIREIA EFARMOGON YPSILIS TECHNOLOGIAS
ENG	ENGINEERING - INGEGNERIA INFORMATICA SPA
UST	UNIVERSITAET STUTTGART
PRQ	PRAQMA A/S

The RADON project (January 2019 - June 2021) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No

Executive summary

As part of the RADON framework, the Defect Prediction (DP) tool focuses on Infrastructure-as-Code (IaC) correctness. It consists of several components to mine open-source repositories to extract quality metrics and features to guide the empirical training and enrichment of the model for defect prediction, and finally predicting possible code smells and errors in IaC blueprints. The Defect Prediction is envisioned as a tool which is: (i) agnostic to IaC technologies, and; (ii) specific to address certain IaC defects, with a focus on code smells. Currently, it targets particularly Infrastructure-as-Code templates and blueprints in Ansible and will be later extended to TOSCA blueprints. This document describes the first version of the defect prediction tool. The main contributions are (i) a catalogue of IaC-oriented metrics used to train and validate the built models and (ii) a novel standalone Machine-learning based tool suitable to classify defect-prone infrastructure scripts. The tool consists of three artefacts, namely a *crawler* to collect relevant open-source IaC repositories, a *repository miner* to identify existing defect-free and defect-prone scripts and a *defect prediction* component to continuously/periodically build a Machine-learning classifier to detect defect-prone scripts. All DP-related artifacts described in this document are publicly available on GitHub[1].

[1] Crawler and repository miner are available at: https://github.com/radon-h2020/iac-miner
The Defect Prediction component is available at: https://github.com/radon-h2020/radon-defect-prediction-api

Glossary

DP	Defect Predictor
IaC	Infrastructure-as-Code

Table of contents

1. Introduction

IT infrastructures are constantly evolving and growing in size and complexity but still little is known concerning how to best maintain, speedily evolve, and continuously improve the code behind the IaC practice and yet it is picking up more and more traction in different domains. This is problematic for organizations where software is essential. Infrastructure failures are even more demanding in environments where IT systems are more than just business-critical and where there is no tolerance for downtime. For example Amazon's systems handle hundreds of millions of dollars in transactions every day. In that context, only in 2012 the estimated average cost of one-minute service downtime for Amazon alone was $66,000[2], even after extensive manual and semi-automatic service continuity practices such as service hot stand-by, or elastic provisioning.

Software Defect Prediction is one of the most assistive activities of the testing process across the software development and operations life-cycle. This technology activity identifies the parts of the systems that are defect-prone and require more extensive testing. As any other source code artifact, infrastructure configuration management files scripts can be defect-prone. However, in contrast to the several defect-prediction tools proliferating for General Purpose Languages, the state-of-the-art of Infrastructure-as-Code lacks quality assurance tools to such an extent that industrials explicitly mentioned the need for instruments to support them when developing infrastructure code [Gue2019].

Therefore, definition of effective prediction of defect-prone IaC scripts enables such Organization and DevOps to focus on critical scripts during Quality Assurance (QA) activities, and allocate effort and resources more efficiently to ensure high-quality and more correct infrastructure code allowing for costs savings in infrastructure maintenance costs as well as erroneous orchestration failure and costs connected to that.

This deliverable aims at describing a software defect prediction tool for IaC to help software practitioners in prioritizing their inspection efforts for IaC scripts by proposing prediction models of defective IaC scripts and investigating the role of code and process metrics for their prediction. To this end, this document describes the RADON Framework for IaC Defect Prediction, a fully integrated machine-learning-based framework that allows for repository crawling, metrics collection of 92 code and 16 process metrics, model building and evaluation. The RADON Defect Predictor will, in essence, allow savings in costs by (1) instrumenting more focused maintenance and testing as well as by (2) avoiding erroneous execution risks which can lead to hazardous or even harmful infrastructure failures. More specifically, the RADON defect prediction tool is well beyond the state of the art and practice since neither offer solutions for infrastructure code automated maintenance (e.g., the focused maintenance facilities allowed by our defect prediction solution) nor solutions for predictive orchestration error resolution, which are enabled by the

[2] As determined by Forbes based on Amazon's 2012 net sales.

RADON defect prediction tool as a first-of-its-kind solution for infrastructure code maintenance and evolution.

1.1 Deliverable objectives

The main objective of the deliverable is to document the first version of the DP architecture and implementation, provide examples of its usage, while highlighting its main achievements so far.

This objective can be broken down into the following parts that are reflected in the structure of this deliverable:

- An overview of the requirements that drive the development of DP and an overview of the proposed KPIs, as described in the previous deliverables [D2.1], [D2.2.] and [D6.1].
- A description of the metrics employed to characterize IaC code quality. These metrics are employed in the RADON defect prediction models to detect defect-prone blueprints.
- A description of the architecture and implementation of the RADON Framework for IaC Defect Prediction. In this regard, the deliverable provides (i) a general overview of the framework and its components, (ii) a detailed description of their architecture, and (iii) the workflow of the user interaction.
- A step-by-step illustration of the DP APIs usage and of the tools it relies on (developed in the scope of RADON) for crawling open-source repositories, mining them and extracting software metrics, building the Machine-Learning models and evaluating them.

1.2. Overview of main achievements

This main achievements reported in this deliverable reflect that it is already possible to

- crawl open-source repositories from Github;
- mine defect-prone and defect-free instances from the collected repositories to build classifiers for the detection of defective IaC blueprints;
- train those classifiers and use them for defect prediction;
- graphically interact with the models through a plugin for Visual Studio Code and Eclipse Che, integrated in the RADON IDE.

1.3 Structure of the document

The remainder of this document is organized as follows. Section 2 outlines usage scenarios and the requirements of the DP. Section 3 describes in detail a catalogue of metrics to characterize code quality of IaC that is the core of the defect prediction functioning. Section 4 illustrates the architecture of the DP and the related tools, while Section 5 shows examples of usage of these tools. Finally, Section 6 concludes the document.

2. Requirements

2.1 Actors and external stakeholders

The deliverables [D2.1] defines a set of actors intended to be supported by RADON. All actors involved in the development and release activities are relevant for the Defect Prediction tool [D2.3], namely (i) the Software Developer, (ii) the Release Manager, (iii) the Operations Engineer (Ops), and (iv) the QoS Engineer.

2.2 Use cases and usage scenarios

Deliverable [D2.1] defined the use cases for the Defect Predictor, including the interaction with the RADON user and the other RADON tools. The use cases had been grouped into six main usage scenarios, namely (i) defects detection; (ii) model running; (iii) metrics extraction; (iv) language not supported; (v) addition of new defects, (vi) update of predictions, and (vii) model training and testing.

2.2.1 Usage Scenarios

Usage scenarios from [D2.1].

US1	Detect defects	The RADON user starts the analysis on the desired IaC script from the IDE clicking on the appropriate boundary object. The *IDE* loads the blueprint and launches the integrated *defect prediction* tool, passing it the IaC script.
US2	Run model	The defect prediction tool receives an IaC script and runs the model (i.e., a classifier) to predict whether or not the script presents a defect, and if so, returns the class of defect detected.
US3	Extract metrics	The defect prediction tool extracts the metrics (among those implemented in the tool) from the IaC script. Then it passes them to the model to predict the defect-proneness of the script.
US4	IaC language not supported	When the IaC script language is not supported the tool will report an error message to notify the user through the IDE.
US5	Add new defects	This use case illustrates the scenario for the defect prediction tool in which a RADON user may want to update the model(s) with a new example of a defective script, for further automatic detection within the tool. A user manually classifies a script as defective or not (either to fine-tune the predictor or just to add new training data). The IDE dispatches the control to the defect prediction tool which updates the existing training data with the new information. Then, it re-trains the model based on the new training set.

US6	Update prediction	The use case differs from "Add new defects" (US5) as the *user* manually updates the classification to correct it to fine-tune the predictor or adjust the training data.
US7	Train and test model	The defect prediction tool gets an IaC script, extracts metrics from it and stores the relevant information in the training data. Then it trains the model with the augmented training data, tests it, and stores the new model(s) to be used next.

2.3 Requirements table

The following table lists the DP-related requirements as published in the deliverable [D2.1]. However, lessons learned from the evaluation of the first prototype of the defect prediction tool led to two changes in the requirements for the defect prediction tool:

- Updated R-T3.4-5, which states that the defect prediction tool could provide a defect threat level to architecture elements and predict threat-level defects under certain infrastructure assumptions. This is a typical regression problem that requires to identify the number of bugs in the infrastructure to establish a threat-level. However, it requires an ontology of IaC bugs, which does not exist yet. In addition, the current defect predictor is a classification model based on Decision Tree or Random Forest. Therefore, the requirement has been changed to address its explainability by providing the user with a set of rules that identify defective-prone IaC scripts and the decision path that led to the final prediction. This requirement has also been raised by one of the industrial partners and therefore it's priority changed from COULD to MUST HAVE.
- Updated priority of R-T3.4-3, which states that the defect prediction tool *could* provide a command line interface. Although initially not a required functionality, partners from industry, in particular PRQ, requested to work with a command-line interface provided by the defect predictor tool. Therefore, the new priority for the requirement is MUST HAVE.
- Removed R-T3.4-11, which states that the defect prediction tool must decrease the bug-fixing times with respect to manual inspection. First, the requirement partially overlaps R-T3.4-10, which states that the defect-prediction tool must improve performances over manual inspection. Second, the goal of the defect predictor is to locate bug-prone software components and not to automatically fix them. Indeed, bug fixing is a developers' concern and depends on their ability to find a fix for a given bug. As the defect-predictor is essentially a detector, we stick with the requirement R-T3.4-10 to improve the performance in identifying defects during manual inspection.

ID	**Description**	**Priority**
R-T.3.4-1	The defect-prediction tool must provide APIs to be easily integrated with other tools in RADON and with the IDE	Must have
R-T.3.4-2	The defect-prediction tool must display a GUI for the plugin (to be integrated	Must have

	into the IDE)	
R-T.3.4-3	The defect-prediction tool could provide a command-line interface	Must have
R-T.3.4-4	The defect-prediction tool could provide an interactive interface to allow developers and operators to report pieces of the infrastructure code where defects or antipatterns are present	Could have
R-T.3.4-5	The defect-prediction tool must provide a set of rules that identify defect-prone scripts and an interpretation of the final decision	Must have
R-T.3.4-6	The defect-prediction tool must provide built-in functionalities to be able to communicate with infrastructure elements in an automatic fashion	Must have
R-T.3.4-7	The defect-prediction tool must provide a linter to flag programming errors, bugs, style errors, and warnings	Must have
R-T.3.4-8	The defect-prediction tool must provide filters to decide which predefined defects to find	Must have
R-T.3.4-9	The defect-prediction tool must be able to ingest data, also in real-time, from multiple sources	Must have
R-T.3.4-10	The defect-prediction tool must improve performances over manual inspection	Must have

2.4 Key performance indicators

Deliverable [D6.1] defines two key performance indicators (KPIs) for the Defect Predictor, relevant for this deliverable as they impose requirements on functionality:

KPI 1	IaC Languages Coverage	We expect to be able to express and execute at least two (≥2) IaC languages supported as input by the defect predictor. These languages may include Ansible, Chef, Puppet, Tosca, etc. In the context of D6.1 we expected to provide support for Ansible (currently supported) and extend it to the TOSCA orchestration language (ongoing).
KPI 2	IaC Defect Types	It should be possible to identify at least five (≥5) types of IaC defects supported by the defect prediction algorithm. These types may include bugs and bad smells like *God Blueprint, Long Resources, Circular dependencies, etc.*

3. Catalogue of software quality metrics for infrastructure code

The DP tool relies on supervised and unsupervised machine learning techniques. A set of metrics to characterize IaC scripts is needed to feed the models. To this aim, we developed a catalogue of software metrics to describe the structural characteristics of such files. On the one hand, these metrics allow DevOps engineers to model the quality aspects of IaC. On the other hand, they allow for effectively maintaining and evolving the scripts during Quality Assurance activities, by classifying them according to the type of defects required by KPI2.

This section illustrates the catalogue composed of 46 measures that identify quality IaC code properties for Ansible. The advantages of a metrics-based quality management approach to infrastructure code are manifold, among others:

- Source code properties can be used as *early indicators of faulty infrastructure scripts* potentially leading to expensive infrastructure failures;
- The analysis of IaC properties can *help developers to understand and improve the quality of their infrastructure* through incremental refactoring, as opposed to the conventional trial-and-error approach;
- Specific metrics can be defined across IaC languages (e.g., TOSCA) to understand the mutual and combined characteristics of Infrastructure-as-Code blends as opposed to focusing on a single vendor-locked IaC solution.

The catalogue has been built by first looking for traditional and language-agnostic source code metrics that are potentially applicable to IaC, stemming from a survey of almost 300 traditional and object-oriented source code metrics [Nun2017], such as executable, commented and blank lines of code, function count, class entropy complexity, and average method size. Needless to say that most of the object-oriented metrics do not apply to IaC, and at the moment 8 are present in the catalogue.

Then, some of the metrics applicable to IaC were introduced by previous work on Puppet [Rah2019] and ported to Ansible.

Finally, we searched for metrics that are specifically inherent to IaC scripts written in Ansible, starting from the *atomic* structural characteristics described in the documentation [AnsDoc] for which structural metrics were directly implementable, and moved towards the more complex ones that spread through multiple scripts and/or can be expressed as a combination of atomic measures. hose cover most of the Ansible constructs (e.g., plays, tasks and modules), and include metrics dealing with error handling, bad and best practices (e.g., using deprecated statements and naming tasks uniquely, respectively), access of data from outside sources, and more.

Afterwards, it was possible to classify the initial set of metrics in three groups: (i) object-oriented metrics that can be ported to Ansible, and IaC in general; (ii) metrics that were investigated in previous works on IaC and that can be ported to Ansible, and therefore to similar languages; and

(iii) metrics related to best and bad practices in Ansible (which are often reported in the documentation or in external resources as books).

The first two sets concern metrics that were investigated with respect to their value to code quality (even though some of them not yet studied in the context of infrastructure code); the latter set emerged when analyzing the recommendations to design quality infrastructure code.

Overall, the catalogue is composed of 46 code metrics. In particular, (i) 8 metrics are related to traditional ode metrics and reflect language-agnostic code characteristics, (ii) 14 metrics have been adapted by those previously developed by [Rah2019] for Puppet, (iii) 24 metrics concerns some inherent characteristics of Ansible that are observable in other orchestration configuration languages as well.

3.1 Traditional code metrics

The metrics related to the first group (object-oriented) are discussed in the following. They all concern the characterization of long/complex infrastructure code: as widely reported in the literature on source code quality [Dam2012], [Zha2009], those metrics could potentially make the code more prone to be defective. While no empirical evaluation of the impact of these metrics is still available in the context of IaC, we hypothesize that similar conclusions could be reached.

- ***LinesSourceCode***, ***LinesComment***, and ***LinesBlank*** to count the total number of *executable lines of code*, *lines of comments*, and *blank lines*, respectively.
- ***NumConditions*** and ***NumDecisions*** where a *condition* is a Boolean expression containing no Boolean operators (e.g., 'and' and 'or') and a *decision* a Boolean expression composed of conditions and one or more Boolean operators.
- ***TextEntropy*** to measure the complexity of a script based on its information content, analogous to the *class entropy complexity*.
- ***NumTasks*** to measure the number of functions in a script, analogous to the traditional *Number of Methods Call* [Nun2017]. An Ansible *task* can be considered equivalent to a method, as its goal is to execute a module with very specific arguments.
- ***AvgTaskSize*** analogous to the traditional *Average Method Complexity* [Nun2017], to measure the average size of program modules.

3.2 Code metrics ported from previous works on IaC

Follows the second group of metrics that we generalized from the previous work conducted by [Rah2019] where the authors observed a significant correlation with defective infrastructure as

code scripts. Specifically, they conducted a qualitative analysis with practitioners and empirically validated such metrics in the scope of defect prediction of Puppet code:

- ***NumCommands*** - Puppet allows developers to execute external commands via the resource type *exec*. For the same functionality, Ansible provides several modules: *command*, *expect*, *psexec*, *raw*, *script*, *shell*, and *telnet*.
- ***NumEnsure*** - *ensure* is a Puppet source code property used to check the existence of a file, directory or symbolic links. In Ansible the existence of those entities can be checked through the module *stat*.
- ***NumFile*** - *file* is a source code property used to manage files, directories, and symbolic links. It exists either in Puppet (as a resource type) and Ansible (as a module).
- ***NumFileMode*** - *mode* is a source code property used to set permissions of files. It exists either in Puppet (as an attribute of the *file* resource type) and Ansible (as a parameter of the *file* module).
- ***NumInclude*** - In Puppet, other scripts can be executed with the *include* function. This functionality in Ansible is provided by several *include* and *import* modules that allow users to break up large playbooks into smaller files, which can be used across multiple playbooks or even multiple times within the same playbook. Import statements are pre-processed at compilation-time:
    - ***NumImportPlaybook*** - *import_playbook* is used to include a file with a list of plays to be executed in the current playbook;
    - ***NumImportRole*** - *import_role* is used to load a *role* when the playbook is parsed;
    - ***NumImportTasks*** - *import_tasks* is used to import a list of tasks to be added to the current playbook for subsequent execution.

  Include statements are processed at execution-time:
    - ***NumInclude*** - *include* is used to include a file with a list of plays or tasks to be executed in the current playbook;
    - ***NumIncludeRole*** - *include_role* is used to dynamically load and a specified role as a task;
    - ***NumIncludeTasks*** - *include_task* is used to include a file with a list of tasks to be executed in the current playbook.
    - ***NumIncludeVars*** - *include_vars* is used to load YAML or JSON variables from a file or directory, recursively, during task run-time.

- **NumParameters** - In Puppet the state of a resource is described with an attribute. Similarly, Ansible *parameters* (or arguments) describe the desired state of the system.

- ***NumSSH*** - *ssh_authorized_key* is a Puppet source code property used to manage SSH authorized keys. The analogue in Ansible is the module *authorized_key*, used to add or remove SSH authorized keys for particular user accounts.

- ***NumURLs*** - URL refers to URLs used to specify a configuration. Ansible defines a module called *uri* to interact with *http* and *https* web services, and requires to set a parameter url.

3.3 Ansible-based metrics

The third group of metrics was derived from the Ansible documentation and are mainly related to best and bad practices and (external) data management. These metrics could potentially affect the quality of infrastructure code both in terms of comprehensibility and maintainability:

- ***DeprecatedKeywords*** and ***DeprecatedModules*** - Deprecated modules and keywords usage is discouraged as they are kept for backwards compatibility only.

- ***NumBlocks*** and ***NumBlocksErrorHandling*** -- A *block* logically groups tasks within a section, but also allows for exception handling by appending a *rescue* or an *always* to the block. The tasks in the block are normally executed. A *rescue* section is executed only when an error is raised, while an *always* section is executed in any case (i.e., included in case of errors).

- ***NumDistinctModules***, ***NumExternalModules***, and ***NumFactModules*** - Modules are reusable and standalone scripts called by tasks. Many modules are maintained by the community. However, users can create and maintain their modules, called *external modules*. Furthermore, some modules do not change the state of the system but only return data: they are called *fact modules*. It is worth noting that the modules maintained by the community are fully documented and tested, while this is optional for external modules. Therefore, we conjecture that a blueprint with the latter modules is more difficult to maintain than a blueprint containing the former. At the same time, we hypothesize that blueprints with many fact modules are less prone to unexpected behaviours and easier to test, as they do not alter the state of the system. Finally, we hypothesize that a blueprint consisting of many distinct modules is less self-contained and potentially affect the complexity and maintainability of the system, as it is responsible to execute many different tasks rather than a task several times, with different options, for example ensuring the presence of dependencies in the system.

- ***NumFilters*** - Filters are used to transform data inside a template expression, such as formatting data or rendering them in a different format, forcing variables to be defined (i.e.,

the default behaviour from Ansible is to fail if variables are undefined, but this aspect can be turned off through filters) or defaulting undefined variables, omitting module parameters, combining dictionaries and more. Filters can be concatenated to perform a sequence of actions. While they allow for transforming data through a sequence of actions in a very compact way, we believe filters may potentially affect the readability and maintainability of the code.

- ***NumIgnoreErrors*** - Ansible provides different ways to handle errors. Among others, it is possible to prevent a playbook from stopping when a task fails by setting *ignore_errors: True*. However, ignoring errors is considered as a bad practice, since *ignore_errors* hides error handling.

- ***NumLookups*** - Lookups are an advanced feature that allows access to outside data sources, and require a good working knowledge of Ansible plays before incorporating them. Some lookups pass arguments to a shell, and one should use them carefully to ensure safe usage.

- ***NumSuspiciousComments*** - Suspicious comments warn the presence of defects, missing functionality, or weakness of the system.

- ***NumUniqueName*** - naming plays and tasks uniquely is a best practice to quickly locate problematic tasks. Duplicate names may lead to not deterministic or at least not obvious behaviours [Kea2015].

- ***NumNamesWithVariables*** - Having uniqueness as a goal, many playbook developers prefer to use variables instead of hard-coding names. This strategy may work well but authors need to take care of the source of the variable data they are referencing. Variable data can come from a variety of locations, and the values assigned to variables can be defined at a variety of times. For the sake of play and task names, only variables for which the values can be determined at playbook parse time will parse and render correctly. If the data of a referenced variable is discovered via a task, the variable string will be displayed unparsed in the output [Kea2015], potentially affecting debugging and software auditing.

- ***NumUserInteractions*** - In some cases, an Ansible script requires the user input (e.g., username and password to access a service). Asking for external input may potentially affect the correctness of the program at run-time. User interactions have to be handled by the program with several conditions. We conjecture that, if not handled properly, a given input may lead the system to crash at run-time.

The remaining metrics are self-describing and measure different aspects of the size of a blueprint which may affect its quality in terms of complexity and readability: ***NumPlays***, ***AvgPlaySize***, ***NumRoles***, ***NumVariables***, ***NumLoops***, ***NumMathOperations***, ***NumPaths***, ***NumRegex*** (i.e., regular expressions), ***NumTokens*** (i.e., words separated by blank spaces), and ***NumKeys*** (i.e. keys

of the dictionary representing a playbook or a list of tasks). In particular, paths and regular expression are often subject to typos, which might lead to run-time errors if they are not properly handled. We conjecture that the more they are, the higher the chance the system will run into unexpected behaviour. In general, we hypothesize that the higher the number of the aforementioned source code properties, the more complex the blueprint.

4. Defect Prediction tool architecture and implementation

The tool consists of three artefacts, namely a *crawler* to collect relevant open-source IaC repositories, a *repository miner* to identify existing defect-free and defect-prone scripts and a *defect prediction* component to continuously/periodically build a Machine-learning classifier to detect defect-prone scripts. This section describes the general tool architecture and its components in detail.

4.1 Framework overview

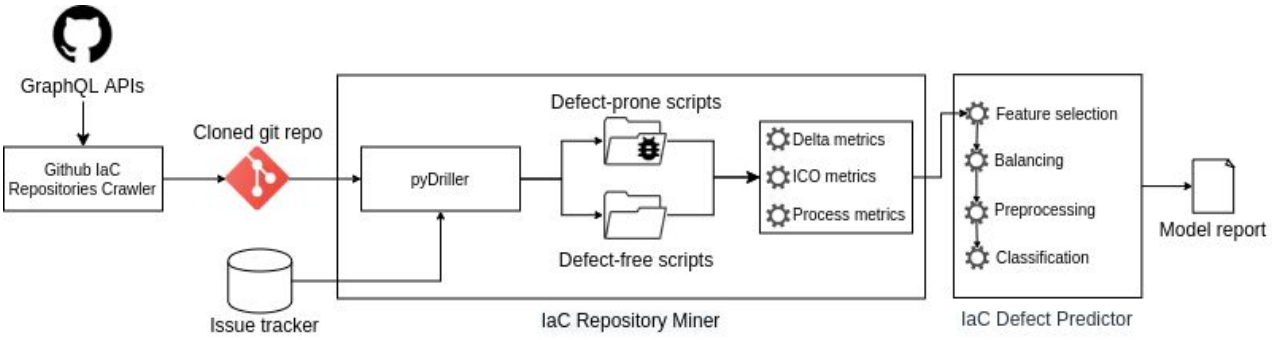


Figure 1 - The RADON Framework for IaC Defect Prediction

Figure 1 provides an overview of the Defect Prediction tool architecture for defect prediction consisting of 3 main components: (i) ***Github IaC Repositories Crawler***, (ii) ***IaC Repository Miner***, (iii) ***IaC Defect Predictor***.

The *Github IaC Repositories Crawler* collects active and relevant repositories on GitHub.

The *IaC Repository Miner* mines defect-prone and defect-free IaC scripts from a repository. Then, it gathers a broad set of metrics from the literature comprising of traditional application code metrics (e.g. lines of code), IaC-oriented metrics (e.g., number of configuration tasks), and process metrics (e.g. number of commits to a file), which are computed upon the collected IaC scripts to predict their defect-proneness.

Finally, the *IaC Defect Predictor* pre-processes the datasets and trains the Machine Learning models. Given an unseen IaC script, this component classifies it as *defect-prone* or *defect-free*.

4.1.1 Github IaC Repositories Crawler

The *Github IaC Repositories Crawler* searches for *public* candidate repositories containing Ansible code through the novel GraphQL-based GitHub APIs[3]. The tool crawls for public repositories

[3] https://developer.github.com/v4/

created since 2014. Ansible has been developed since 2012 and we assume that two years is a reasonable amount of time for a new language to gain popularity.

To mine only active and relevant projects, the *Github IaC Repository Crawler* verifies several constraints which are reported in **Table 3.6 - criteria** along with their description and rationale.

#	Name	Description	Rationale
1	# push events	The repository must have at least one *push event* to its default branch in the last six months	Evidence of recent development activity
2	# releases	The repository must have at least 2 releases	The proposed defect predictor analyzes files at each release and between successive releases
3	Ratio of IaC scripts	At least 11% of the files must be IaC scripts	Repositories must have a sufficient IaC scripts
4	# of Core Contributors	The project must have at least 2 core contributors	Evidence of collaboration
5	Continuous Integration	The repository must rely on continuous integration	Evidence of quality
6	Comment Ratio	The *comment ratio* must be at least 0.2%	Evidence of maintainability
7	Commit Frequency	The *commit frequency* must be at least 2.0	Evidence of project evolution
8	Issue Frequency	The *issue frequency* must be at least 0.023	Evidence of project management evolution
9	License Availability	The repository must have evidence of a *license*	Evidence of accountability evolution
10	# Lines of Code	The repository must have at least 190 *lines of code*	Co-assess and control the criteria 4-9

Table 3.6 - criteria

Criterion 1 allows the crawler to discard inactive projects, while criterion 2 is needed because the target models are trained at release-level. The *ratio of IaC Scripts* represents a cutoff to analyze

repositories containing IaC scripts that has been determined by the previous works ([Jia2015], [Rah2018], [Rah2019]), whereas the criteria 4-10 are considered as good indicators of well-engineered software projects [Mun2017]. These criteria are also used to find the most similar projects to get a pre-trained model to be used when a given project has not enough history information to train a new model from scratch.

Repositories which are archived, disabled, mirrored, or forked are excluded as well.

4.1.2 IaC Repositories Miner

IaC Repository Miner analyzes the history of the projects and extracts the *defect-prone* and *defect-free* blueprints needed for the analysis. It works at release level of granularity: for each software release a new dataset is created. To this end, first it applies the following process to identify *defect-fixing* commits:

1. it extracts the commits linked to issues closed and related to bugs (i.e., with labels 'bug', 'bugfix', etc). Please consider that GitHub provides an issue tracker that links commits and corresponding issue reports;

2. it analyses the commits whose messages indicate defective scripts. Specifically, when analyzing the commits messages, it removes all words ending with 'bug' or 'fix', since those terms can be affixes of other words as "debug" and "prefix". Then, as previously done by Zhang et al. [Zhang14], it labels as defect-fixing the commits that match the following regular expression:

```
(bug | fix | error | issue | crash | problem | fail | defect | patch)
```

The *IaC Repository Miner* keeps only the commits that modify at least one Ansible script. Afterwards, it determines their defect-proneness as follows. For each Ansible script in a *defect-fixing* commit, it relies on the *SZZ* algorithm [kim2006] to automatically identify the **first** commit that introduced a defect in that script. Then, it labels as *defect-prone* all the snapshots of a file between its *first defect-inducing commit* (inclusive) and the defect-fixing commit. Finally, it selects all the releases containing *at least one defect-prone* script.

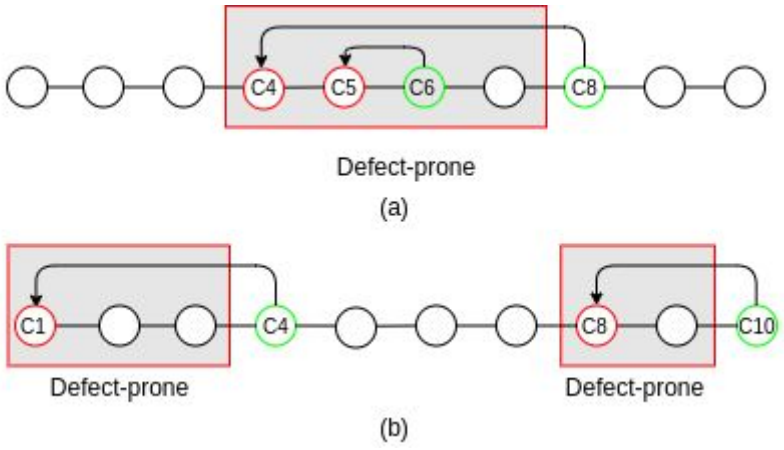


Figure 2 - Two scenarios of the labelling process.

Figure 2 depicts two possible scenarios for the labelling process. In both cases, several defect-fixing (green circles) and defect-inducing (red circles) commits modify a file in chronological order.

In **Figure 2a**, although commit C6 fixes a defect introduced in C5, C8 fixes a defect introduced in C4 (i.e., before the previous fix). This means that C6 is only a partial fix and the file is still defect-prone. Therefore, the file is labeled as *defect-prone* at commits C4, C5, C6 and C7 (gray box).

In **Figure 2b**, C10 fixes a defect introduced in C8, later than the previous fix in C4. Consequently, the file is labelled as *defect-prone* from commit 1 to 3 and from commit 8 to 9. The other snapshots of the file (i.e., outside the grey boxes) are labelled as *defect-free*.

Along with the bug-proneness of the IaC scripts, *IaC Repository Miner* gathers a comprehensive set of 108 features to train the defect prediction model.

Such features have been extracted from previous work in defect prediction ([Mos2008], [Ari2010], [Rah2013], [Dal2020]) and can be classified into three categories:

- **IaC-oriented metrics (ICO)** (described in Section 3) are structural properties derived from the source code of infrastructure scripts. *IaC Repository Miner* uses the 46 metrics proposed by [Dal2020] that include the number and the size of plays and tasks, the number of commands, best and bad practices, and more.

- **Delta metrics** capture the amount of change in a file between two successive releases. Such metrics are collected for each IaC-oriented metric as previously done for defect prediction by [Ari2010].

- **Process metrics** capture aspects concerning the development process rather than the code itself ([Mos2008], [Rah2013]). *IaC Repository Miner extends* PyDriller[4] to collect such measures that include the number of developers that changed a file, the total number of added and removed lines, the number of files committed together.

4.1.3 IaC Defect Predictor

The *IaC Defect Predictor* builds the pipeline that balances and pre-processes the dataset, trains and validates the Machine-Learning models, and uses it to predict unseen instances. It uses different configurations in terms of feature selection, normalization, data balancing, classifiers and hyper-parameters.

1. **Feature selection**. Not always the data is originally intended for defect prediction, therefore not all features in the dataset may be helpful for the task, because they are constant or do not provide useful information exploitable by a learning method for a particular dataset. The RADON framework uses feature selection to reduce the size of the dataset and speed-up the training, but also to select the optimal number of features that maximize a given performance criterion.

2. **Data balancing**. Once feature selection is finished, the training data are balanced such that the number of defect-prone instances equal the number of defect-free instances. The RADON framework uses three configurations for balancing, namely (i) no balancing; (ii) random under-sampling the majority class; and (iii) random over-sampling the minority class.

3. **Data normalization**. In this step the training data are normalized scaling numeric attributes. The RADON framework uses three configurations for data normalization, namely (i) no normalization; (ii) *min-max* transformation to scale each feature individually in the range [0, 1]; (iii) *standardization* of the features by removing the mean and scaling to unit variance.

4. **Classification**. The normalized data and the learning algorithm are used to build the learner. Before the learner is tested the original test data are normalized in the same way and the dimensionality is reduced to the same subset of attributes from step 1. After comparing the predicted value and the actual value of the test data, the performance of one *pass* of validation is obtained. Note that in our framework the classification step can be applied with any machine learning algorithm, i.e., the selection of the learner is left to the user.

The final output consists of a *json* file that reports the results and metadata for each validation step and it is accessible to the user.

[4] From release 1.13. See the docs at https://pydriller.readthedocs.io/en/latest/processmetrics.html

4.2 User interaction workflow

In order to analyze the correctness of a blueprint, the user is supposed to train a model using the RADON Defect Prediction Framework or using a pre-trained model via the RESTFull APIs provided by the Defect Prediction tool. In the former case, the user can download the containerised tool or call the API to train a new model. If done offline (i.e., on the user machine) the user can retain all the information about the model. When using the online APIs, public information about the repository and the model will be stored on the server's storage to make the model available for future projects requiring it. If the user decides to use a pre-trained model, it can obtain it from the proper API call, and then use it depending on its needs.

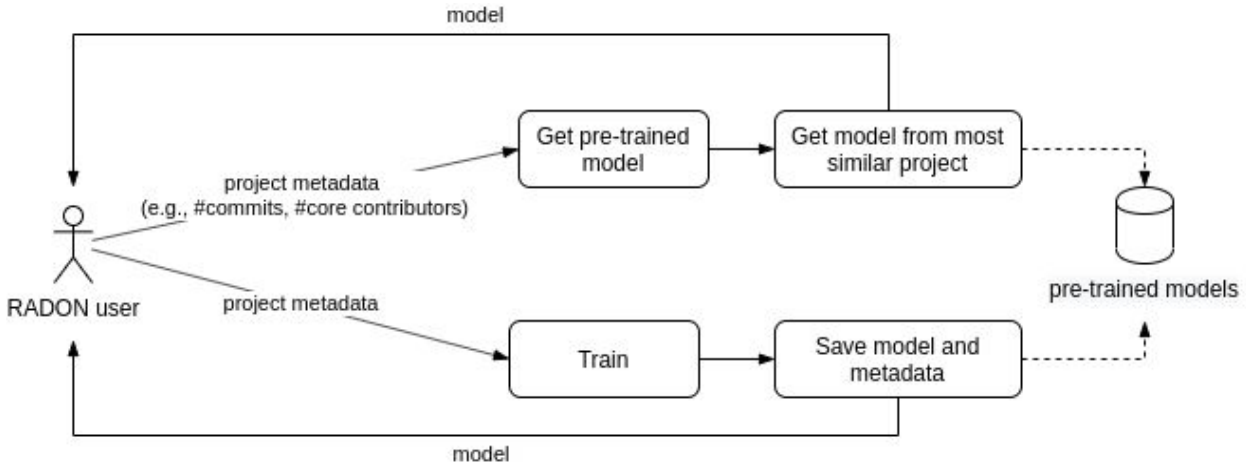


Figure 3 - Simple representation of the aforementioned user interaction workflow.

4.3 Architecture View

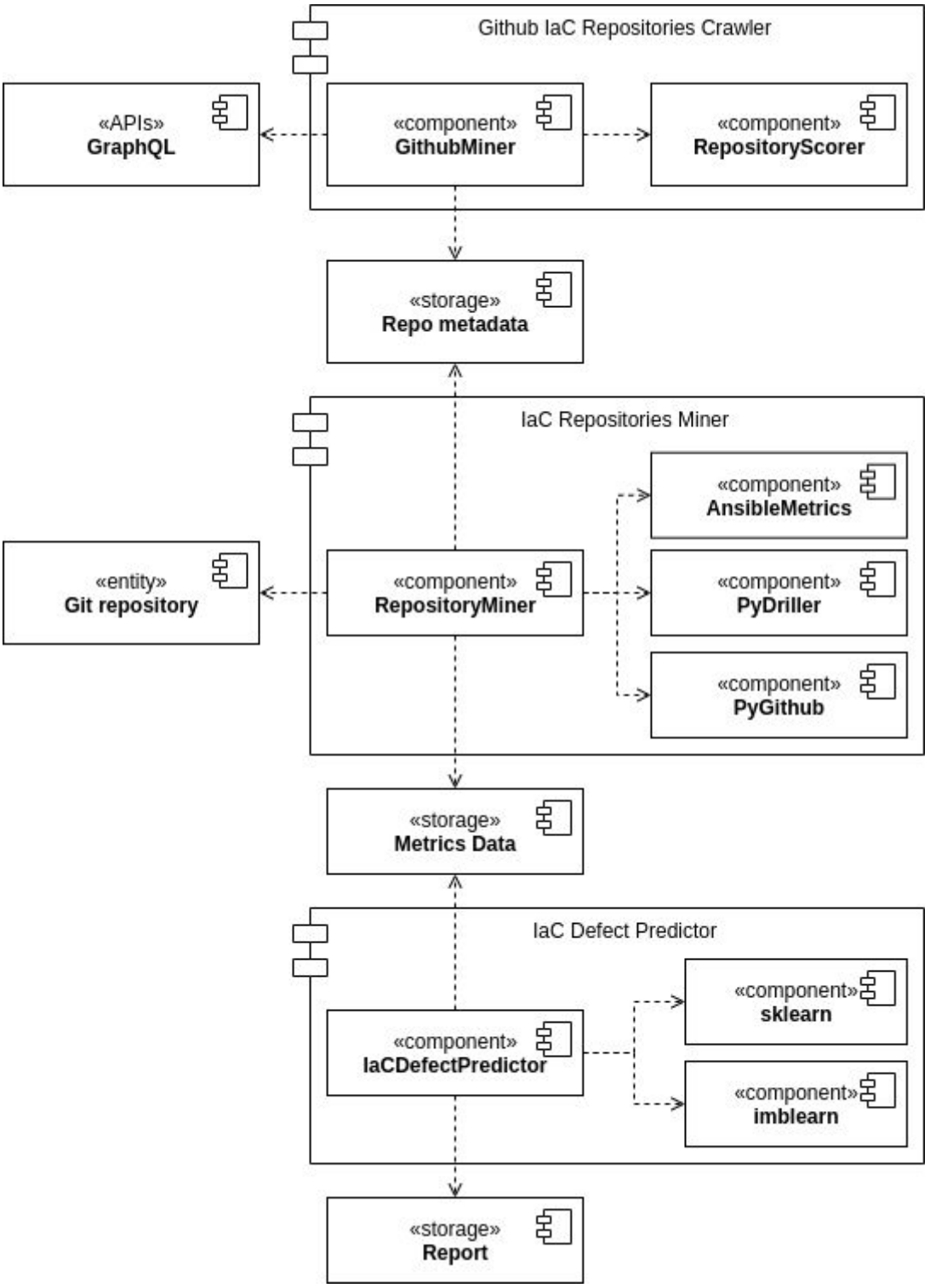


Figure 4 - UML Component diagram of the RADON framework for IaC Defect Prediction

Figure 4 depicts the big picture of the components present in the RADON framework for IaC Defect Prediction using the UML notation. The components are described in detail below.

4.3.1 Github IaC Repositories Crawler

From an architectural perspective, the *Github IaC Repositories Crawler* consists of two components (*GithubMiner* and *RepositoryScorer*) and relies on a third, external component (*GraphQL*). The *GraphQL APIs* are used by the *GithubMiner* component to query Github for public repositories created in a given time frame and pushed after a certain date (*date_from*, *date_to*, *pushed_after* of the class diagram depicted in **Figure 5**) through the method *run_query*. Afterwards, *Github Miner* uses the method *filter_repositories* to filter out repositories based on the user-defined criteria (e.g., minimum number of stars, issues, primary language, etc.).

Afterwards, the *GithubMiner* uses the *Repository Scorer* component to compute the scores related to the criteria in **table 3.5 - criteria**.

It then generates a json object for each collected repository, containing metadata (such as owner, name, url, etc.) and scores, and yields it to the caller. Here, the *GithubMiner* component envisions two main usage scenario, as illustrated later on in section 5.2:

1. The caller can call the method *GithubMiner.mine()* and wait for all the repositories to be collected, before proceeding with the analysis of every repository.
2. The calles can call the method *GithubMiner.mine()* and run the analysis for each repository at a time.

This is done to leave the caller to decide how to employ the tool. The first usage scenario might be faster than the second, but it consumes APIs quotas sooner as well.

4.3.2 IaC Repositories Miner

The IaC Repositories Miner relies on the Python frameworks *PyDriller* and *PyGithub* to analyze the history and the issues of a repository, respectively. It then uses *AnsibleMetrics* to extract product metrics described in Section 3 from the collected IaC scripts. Given the metadata of a repository (RepositoryMetadata in Figure 4) it clones it (*GitRepository* in Figure 4), collects and labels IaC scripts by analyzing the repository's commits and issues with the component *RepositoryMiner* analyze combined

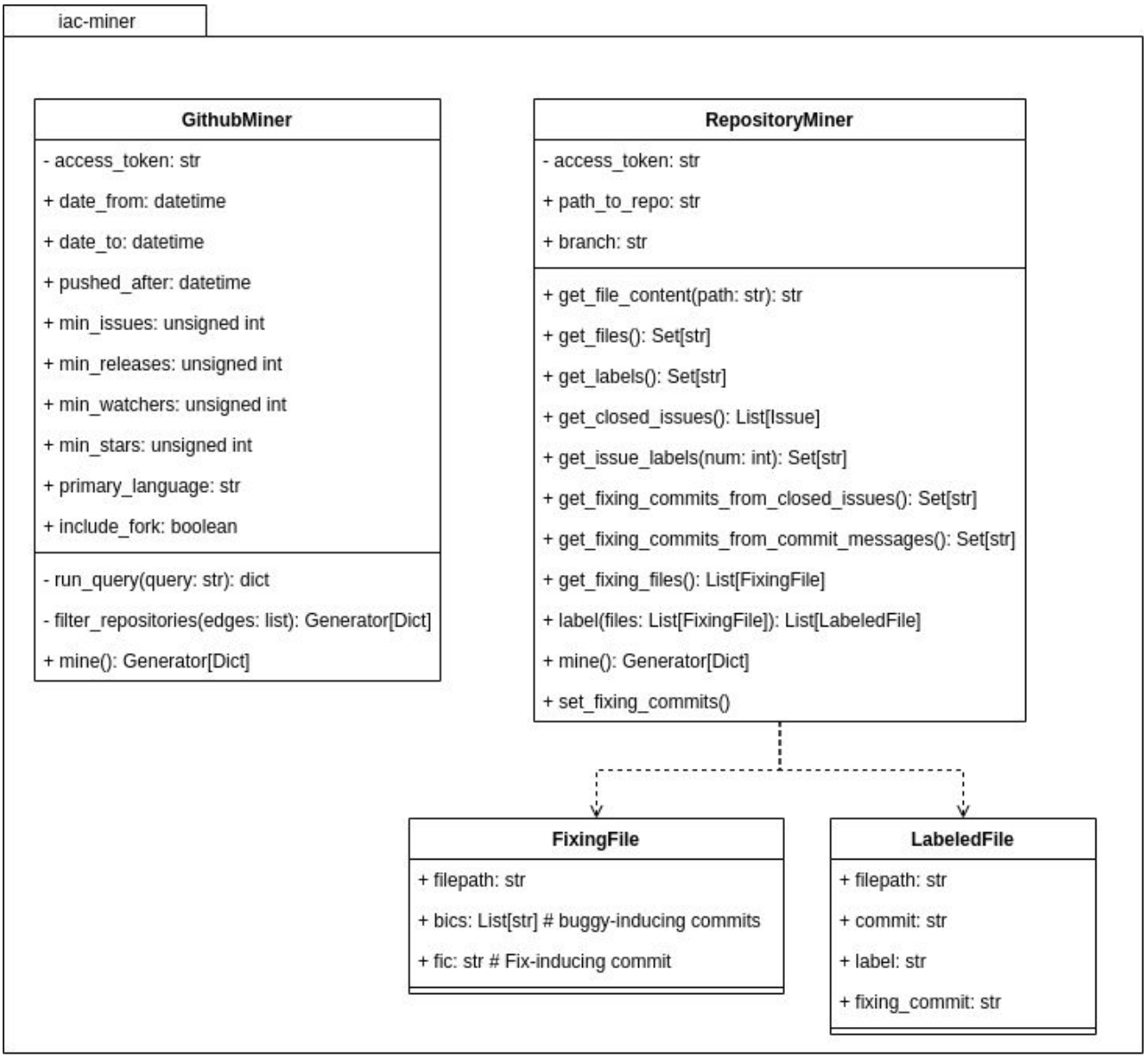
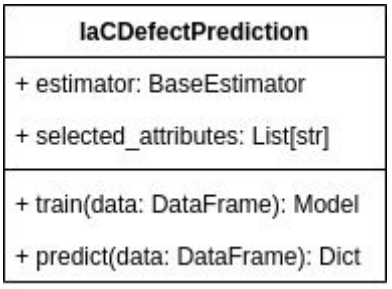


Figure 5 - UML class diagram of the main classes of the *Github IaC Repositories Crawler* and *IaC Repository Miner* components

4.3.3 IaC Defect Predictor

The *IaC Defect Predictor* relies on the Python frameworks *scikit-learn*[5] and *imblearn*[6] to build the pipeline that balances and pre-processes the dataset, trains and validates the Machine-Learning models, and uses it to predict unseen instances.



The DP provides two main endpoints of interaction, described in the following table.

REST Verb	Endpoint	Description
POST	*api/classification/classify*	Classify a YAML-based Ansible script
POST	*api/models/pre-trained-model*	Get a pre-trained model

[5] https://scikit-learn.org/stable/
[6] https://imbalanced-learn.readthedocs.io/en/stable/index.html

5. Demonstration of use

5.1 Outline of the demonstration

The following sections illustrate examples of usage of the components used to collect data to train a defect predictor of IaC scripts and the usage of the Defect-Prediction APIs themselves. The examples are not placed in a specific context. Rather, the goal is to explain and show the general usage of the tool, that is: how to search for relevant IaC-based repositories on Github and mine them to collect data (ie. metadata and metrics for defect-prone and defect-free scripts) to pass to a Machine-Learning classifier for its training and validation.

5.2 Github IaC Repositories Crawler

First, download and install the tool from PyPI (https://pypi.org/project/iacminer/):

```
pip install iacminer
```

Alternatively, download the source code from GitHub:

```
git clone https://github.com/radon-h2020/radon-iac-miner.git
```

and install it locally with:

```
cd iac-miner/
pip install .
```

The tool is ready to use!

5.2.1 APIs usage

The following code snippet shows an example of usage of the GithubMiner APIs that allows to crawl Github and filter repositories based on several criteria like the date of creation, minimum number of releases, primary language, etc.

```
import os
import pandas as pd
from datetime import datetime
from iacminer.miners.github import GithubMiner

miner = GithubMiner(access_token=os.getenv('GITHUB_ACCESS_TOKEN'), #  or hard-code access token, e.g., 'acYrt2....'
                    date_from = datetime.strptime('2020-01-01 00:00:00', '%Y-%m-%d %H-%M:%S'),
                    date_to = datetime.strptime('2020-06-12 00:00:00', '%Y-%m-%d %H-%M:%S'),
                    push_after = datetime.strptime('2020-03-01 12:00:00', '%Y-%m-%d %H-%M:%S'),
                    min_issues = <int>, # default: 0
                    min_releases = <int>, # default: 0
                    min_stars = <int>, # default: 0
                    min_watchers = <int>, # default: 0
                    primary_language = <str|None>, # e.g., python, java, etc. (default: None)
                    include_fork = <True|False>)

for repository in miner.mine():
    print(repository)
    # Clone repository
    # Filter out non-relevant repositories
    # Miner repository (see examples below)
    # Etc.
```

5.2.2 Command-line usage

The tool can be used standalone through a command-line interface via the following command:

```
iac-miner mine-repository --help

usage: iac-miner mine-github [-h] [--from DATE_FROM] [--to DATE_TO]
                             [--pushed-after DATE_PUSH]
                             [--iac-languages [{ansible,chef,puppet,all}] ...]
                             [--include-fork] [--min-issues MIN_ISSUES]
                             [--min-releases MIN_RELEASES] [--min-stars MIN_STARS]
                             [--min-watchers MIN_WATCHERS]
                             [--primary-language PRIMARY_LANGUAGE] [--verbose]
                             dest  tmp_clones_folder

positional arguments:
dest                  Destination folder where to save results to
tmp_clones_folder     Path to a temporary folder where to clone the repositories for their
                      analysis

optional parameters:

-h, --help                  Show this help message and exit
--from DATE_FROMStart searching from this date (default: 2014-01-01 00:00:00)
--to DATE_TO          Search up to this date  (default: 2020-01-01 00:00:00)
--pushed-after DATE_PUSH
                      Search only repositories with a push event after this date (default
                      2019-01-01 00:00:00)
--iac-languages [{ansible,chef,puppet,all}]
                      Only repositories with this language(s) will be analyzed (default: all)
--include-fork        Whether to include forked repositories (default: False)
--min-issues MIN_ISSUES
                      Minimum number of issues the repository must have to be considered
                      for the analysis (default: 0)
--min-releases MIN_RELEASES
                      Minimum number of releases the repository must have to be considered
                      for the analysis (default: 0)
--min-stars MIN_STARS
                      Minimum number of stars the repository must have to be considered
                      for the analysis (default: 0)
--min-watchers MIN_WATCHERS
                      Minimum number of watchers the repository must have to be considered
                      for the analysis (default: 0)
--primary-language PRIMAY_LANGUAGE
                      The primary language of the repository (e.g. python, java, etc.) (default:
                      None)
--verbose             Whether to output steps and results (default: False)
```

5.3 IaC Repositories Miner

First, download and install the tool from PyPI (https://pypi.org/project/iacminer/):

```
pip install iacminer
```

Alternatively, download the source code from GitHub:

```
git clone https://github.com/radon-h2020/radon-iac-miner.git
```

and install it locally with:

```
cd iac-miner/
pip install .
```

The tool is ready to use!

5.3.1 APIs usage

The following example mines a cloned git repository and extracts metrics about defect-prone and defect-free IaC scripts on a per-release basis.

```
import pandas as pd
from iacminer.miners.repository import RepositoryMiner

miner = RepositoryMiner(token=os.getenv('GITHUB_ACCESS_TOKEN'), #  or hard-code access token, e.g., 'acYrt2....'
                        path_to_repo='path/to/cloned/repository',
                        branch='master'
)

data = pd.DataFrame()

# Mine repository and update the dataset with new metrics
for metrics in miner.mine():
   data = data.append(metrics)

# Save dataset as csv
with open('path/to/csv', 'w') as out_stream:
   data.to_csv(out_stream)
```

However, it is possible to mine a repository for specific information. For example, the following code snippet allows to mine fixing commits (i.e., commits that fix a bug) from the repository's issue tracker and from the commit messages:

```
import pandas as pd
from iacminer.miners.repository import RepositoryMiner

miner = RepositoryMiner(token=os.getenv('GITHUB_ACCESS_TOKEN'), #  or hard-code access token, e.g., 'acYrt2....'
                        path_to_repo='path/to/cloned/repository',
                        branch='master'
)

# Get fixing commits by analyzing issues
for sha in miner.get_fixing_commits_from_closed_issues():
    print(sha)

# Get fixing commits by commit messages
for sha in miner.get_fixing_commits_from_commit_messages():
    print(sha)
```

Alternatively, it is possible to get all the files touched by a fixing commits, label them and obtain a set of defect-prone files that can be used to train the defect predictor as follows:

```
import pandas as pd
from iacminer.miners.repository import RepositoryMiner

miner = RepositoryMiner(token=os.getenv('GITHUB_ACCESS_TOKEN'), #  or hard-code access token, e.g., 'acYrt2....'
                        path_to_repo='path/to/cloned/repository',
                        branch='master'
)

# Get all Ansible files involved in fixing commits
miner.set_fixing_commits()
fixing_files = miner.get_fixing_files()

# Label the previous files as 'defect-prone' or 'defect-free'
labeled_files = miner.label(fixing_files)
for file in labeled_files:
    print(file.filepath, file.label)
```

5.3.2 Command-line usage

```
iac-miner mine-repository --help

usage: iac-miner mine-repository [-h] [--branch REPO_OWNER]
                                 [--owner REPO_OWNER] [--name REPO_NAME]
                                 [--verbose]
                                 path_to_repo dest

positional arguments:
  path_to_repo          Name of the repository (owner/name).
  dest                  Destination folder to save results.

optional arguments:
  -h, --help            show this help message and exit
  --branch REPO_OWNER   the repository's default branch (default: master)
  --owner REPO_OWNER    the repository's owner (default: None)
  --name REPO_NAME      the repository's name (default: None)
  --verbose             whether to output results (default: False)
```

Therefore, the following command

```
iac-miner mine-repository /home/Documents/github/radon-h2020/radon-defect-predictor /home/Documents/results/
```

will generate a file **metrics.csv** in the folder */home/Documents/results* containing the metrics for the defect-prone and defect-free scripts present in the repository *radon-h2020/radon-defect-predictor,* if any.

5.4 IaC Defect Predictor

Download the source code from GitHub:

```
git clone https://github.com/radon-h2020/radon-defect-prediction-api.git
```

Build and run the docker image as follows:

```
docker build -t radon-defect-predictor:latest .
```

```
docker run -p 5000:5000 radon-defect-predictor:latest
```

5.4.1 APIs usage

The first endpoint (*api/classification/classify*) allows to detect defects in a YAML blueprint.

It requires a plain YAML blueprint as input and returns a JSON response consisting of the following fields:

- **defective:** a boolean indicating whether the script has been predicted as defect-prone (True) or defect-free (False);
- **metrics**: a list of (metric, value) pairs.

As an example, given the following YAML:

```
- host: all
  tasks:
  - debug:
      msg: "Hello World!"
```

One can call the endpoint in the following way (or programmatically):

```
curl -X POST "http://localhost:5000/api/classification/classify" -H "accept: */*" -H "Content-Type: plain/text" -d "-host: all\n\ttasks:\n\t- debug:\n\t\tmsg: 'Hello World!'"
```

And get the following response:


```
{
  "defective": false,
  "metrics": {
    "avg_task_size": 2,
    "lines_code": 4,
    "num_external_modules": 1,
    "num_keys": 4,
    "num_plays": 1,
    "num_tasks": 1,
    "num_tokens": 7,
    "text_entropy": 2.81
  }
}
```

The second endpoint (*api/models/pre-trained-model*) allows a user to get a pre-trained model from the most similar project.

Given a json consisting of scores for the following attributes:

```
{
    "commitFrequency": <number>,
    "coreContributors": <integer>,
    "issueFrequency": <number>,
    "percentComments": <number>,
    "percentIac": <number>,
    "sloc": <integer>,
    "releases": <integer>,
    "percentDefects": <number>,
    "commits": <integer>
}
```

It will return a response consisting of the following two properties:

- **defective**: a list of the attributes selected by the pre-trained model during training, that can be used to reduce the dimensionality of data from the client;
- **metrics**: a string representing the serialized pipeline, encoded with the python library jsonpickle==1.4.1. To decode it, the user can import it in code and call jsonpickle.decode(response['model'])

As an example, running the following curl call

```
curl -X POST "http://localhost:5000/api/models/pre-trained-model" -H  "accept: */*" -H  "Content-Type: application/json" -d '{"commitFrequency": 5, "coreContributors": 3, "issueFrequency": 0.04, "percentComments": 25, "percentIac": 70, "sloc": 5000, "releases": 10, "percentDefects": 8, "commits": 340}'
```

will result in the following response.

{

"attributes": ["avg_task_size", "lines_blank", "lines_code", …, "num_uri", "num_vars", "text_entropy"],

"model": "{\"py/object\": \"imblearn.pipeline.Pipeline\", \"py/state\": {\"steps\": [{\"py/tuple\": [\"var\", {\"py/object\": \"sklearn.feature_selection._variance_threshold.VarianceThreshold\", \"py/state\": {\"threshold\": 0, \"variances_\": {\"py/reduce\": [{\"py/function\": \"numpy.core.multiarray._reconstruct\"}, {\"py/tuple\": [{\"py/type\": \"numpy.memmap\"}, {\"py/tuple\": [0]}, {\"py/b64\": \"Yg==\"}]}, {\"py/tuple\": [1, {\"py/tuple\": [46]}, {\"py/reduce\": [{\"py/type\": \"numpy.dtype\"}, {\"py/tuple\": [\"f8\", 0, 1]}, {\"py/tuple\": [3, \"<\", null, null, null, -1, -1, 0]}]}, false, {\"py/b64\": ... A\"}]}}]}, \"_sklearn_version\": \"0.23.1\"}}]}], \"memory\": null, \"verbose\": false}}"

}

The user can simply run the following code statements to load and use the model:

```
import jsonpickle
# Perform APIs call and save the result in variable ‘response’
model = jsonpickle.decode(response['model'])
y = model.predict(X)
```

6. Conclusions

This deliverable presented the current version of the Defect Prediction (DP) by revisiting DP's requirements, giving a detailed overview of the DP's architecture and implementation, as well as a demonstration of usage.

At this stage, the DP supports

- The crawling of open-source repositories from Github.
- The mining of defect-prone and defect-free instances from the collected repositories to build classifiers for the detection of defective IaC blueprints.
- The training of those classifiers and their usage for the prediction.
- A standalone portable plugin for Visual Studio Code and Eclipse Che, which allows users to graphically use the models. The plugin is integrated within the RADON IDE.

For the next (and final) DP deliverable we plan the following works:

- Crawling open-source repositories from different sources (e.g., GitLab[7]).
- Support for continuous feedback of defect-prone/free scripts.
- Extend support to the TOSCA orchestration language as described in KP1.
- Extend the DP to identify different types of defects as described in KPI2.

[7] https://about.gitlab.com/

Appendix: Compliance with Requirements

The following tables summarize the level of DPT's compliance with the requirements at this stage, following the categories introduced in Section 2. The labels specifying the "Level of compliance" are defined as follows:

- **Mnn** (scheduled): the requirement is not achieved by the current version; a level of ✔✔ is planned for month *nn*,
- ✔ (partially-low achieved):
  the requirement is partially-low achieved by the current version,
- ✔✔ (partially-high achieved): the requirement is partially-high achieved by the current version,
- ✔✔✔ (fully achieved): the requirement is fully achieved by the current version.

Requirements List

ID	Description	Priority	Level of compliance
R-T.3.4-1	The defect-prediction tool must provide APIs to be easily integrated with other tools in RADON and with the IDE	Must have	✔✔✔
R-T.3.4-2	The defect-prediction tool must display a GUI for the plugin (to be integrated into the IDE)	Must have	✔✔✔
R-T.3.4-3	The defect-prediction tool could provide a command-line interface	Must have	✔✔
R-T.3.4-4	The defect-prediction tool could provide an interactive interface to allow developers and operators to report pieces of the infrastructure code where defects or antipatterns are present	Could have	M30
R-T.3.4-5	The defect-prediction tool must provide a set of rules that identify defect-prone scripts and an interpretation of the final decision	Must have	✔
R-T.3.4-6	The defect-prediction tool must provide built-in functionalities to be able to communicate with infrastructure elements in an automatic fashion	Must have	✔✔
R-T.3.4-7	The defect-prediction tool must provide a linter to flag programming errors, bugs, style errors, and warnings	Must have	M28
R-T.3.4-8	The defect-prediction tool must provide filters to decide which predefined defects to find	Must have	M24
R-T.3.4-9	The defect-prediction tool must be able to ingest data, also in real-time, from multiple sources	Must have	✔✔
R-T.3.4-10	The defect-prediction tool must improve performances over manual inspection	Must have	✔

References

[D2.1] INITIAL REQUIREMENTS AND BASELINE

[D2.2] FINAL REQUIREMENTS

[D2.3] ARCHITECTURE & INTEGRATION PLAN I

[D.6.1] VALIDATION PLAN

[AnsDoc] Ansible Documentation.

[Ari2010] Arisholm, Erik, Lionel C. Briand, and Eivind B. Johannessen. "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models." Journal of Systems and Software 83.1 (2010): 2-17.

[Dal2020] Dalla Palma, Stefano, et al. "Towards a Catalogue of Software Quality Metrics for Infrastructure Code." arXiv preprint arXiv:2005.13474 (2020).

[Dam2012] D'Ambros, Marco, Michele Lanza, and Romain Robbes. "Evaluating defect prediction approaches: a benchmark and an extensive comparison." Empirical Software Engineering 17.4-5 (2012): 531-577.

[Gue2019] Guerriero, Michele, et al. "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry." *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019.

[Kea2015] Keating, Jesse. Mastering Ansible. Packt Publishing Ltd, 2015.

[Kim2006] Kim, Sunghun, et al. "Automatic identification of bug-introducing changes." 21st IEEE/ACM international conference on automated software engineering (ASE'06). IEEE, 2006.

[Jia2015] Jiang, Yujuan, and Bram Adams. "Co-evolution of infrastructure and source code-an empirical study." 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, 2015.

[Mos2008] Moser, Raimund, Witold Pedrycz, and Giancarlo Succi. "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction." Proceedings of the 30th international conference on Software engineering. 2008.

[Mun2017] Munaiah, Nuthan, et al. "Curating GitHub for engineered software projects." Empirical Software Engineering 22.6 (2017): 3219-3253.

[Nun2017] Nuñez-Varela, Alberto S., et al. "Source code metrics: A systematic mapping study." Journal of Systems and Software 128 (2017): 164-197.

[Rah2013] Rahman, Foyzur, and Premkumar Devanbu. "How, and why, process metrics are better." 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013.

[Rah2018] Rahman, Akond, and Laurie Williams. "Characterizing defective configuration scripts used for continuous deployment." 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2018.

[Rah2019] Rahman, Akond, and Laurie Williams. "Source code properties of defective infrastructure as code scripts." Information and Software Technology 112 (2019): 148-163.

[Zha2009] Zhang, Hongyu. "An investigation of the relationships between lines of code and defects." 2009 IEEE International Conference on Software Maintenance. IEEE, 2009.