Rational decomposition and orchestration for serverless computing

Deliverable 3.7

Defect prediction tool II

Version: 1.0

Publication Date: 31-March-2021

Deliverable Card

Deliverable	D3.7
Title:	Defect Prediction Tool II
Editor(s):	Stefano Dalla Palma (TJD)
Contributor(s):	Dario Di Nucci (TJD), Damian A. Tamburri (TJD)
Reviewers:	Pelle Jakovits (UTR), Mark Law (IMP)
Type:	Report
Version:	1.0
Date:	31-March-2021
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 825040

Executive summary

As part of the RADON framework, the Defect Prediction tool (DPT) focuses on Infrastructure-as-Code (IaC) correctness. It consists of several components to mine open-source repositories to extract quality metrics to guide the empirical training and enrichment of the models for defect prediction and predict code smells and errors in IaC blueprints. The Defect Prediction is envisioned as a language-agnostic tool specific to address certain IaC defects, focusing on code smells.

This document presents the final version of the DPT. In particular, this deliverable (i) outlines the DPT tool's final integration into the RADON workflow and tool ecosystem, and (ii) presents the improvements introduced to the DPT framework since the previous deliverable.

Glossary

CI/CD	**C**ontinuous **I**ntegration/**C**ontinuous **D**elivery
DP	**D**efect **P**redictor
IaC	**I**nfrastructure-**a**s-**C**ode

Table of contents

1. Introduction

Infrastructure-as-Code (IaC) adoption has rapidly increased in recent years to speed up the provision and configuration of infrastructural resources using automation based on software development practices such as version control and automated testing. These practices ensure consistent and repeatable routines for system provisioning and configuration changes. While automatically managing configurations makes the process more efficient and repeatable, new problems can emerge: practitioners frequently change the infrastructure, which inadvertently introduces defects [Rah2018] whose impact might be significant. For example, the execution of a defective IaC blueprint erased the directories of about 270 Wikimedia users in 2017[1].

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 applying extensive manual and semi-automated service-continuity practices such as hot service standby or elastic provisioning.

Because of its novelty, the state-of-the-practice 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].

A few attempts focused on building supervised binary classifiers to predict defect-prone blueprints to help DevOps engineers schedule testing and maintenance activities across the software development and operations life-cycle. An activity referred to as Software Defect Prediction aims to save resources and time by identifying and prioritizing the defect-prone parts of the systems, which require more extensive testing. It is worth noting that IaC scripts are source code artifacts, and as with any other source code artifact, they can be subject to failures.

Typically, the process for building such classifiers consists of generating instances from software archives such as version control systems in the form of software components (e.g., classes and/or methods). An instance is characterized using several metrics (a.k.a., features) extracted from those components, such as the number of lines of code, and is labeled as "defect-prone" or "defect-free". The labeled instances are used to build the ground truth for a machine learning classifier to learn the features that discriminate and predict defects in a specific component.

Nevertheless, in the context of IaC, those efforts were limited to academia. In contrast, this deliverable describes the ***RADON Framework for IaC Defect Prediction***, a machine learning-based cutting-edge solution to help software practitioners prioritize their inspection efforts for IaC scripts by proposing prediction models to identify defective IaC scripts.. This fully

[1] https://wikitech.wikimedia.org/wiki/Incident_documentation/20170118-Labs
[2] As determined by Forbes based on Amazon's 2012 net sales.

integrated machine-learning-based framework allows for repository crawling, metrics collection for the Ansible and Tosca languages, model building, and evaluation. Ultimately, it will, in essence, enable savings in costs by (1) instrumenting more focused maintenance and testing and (2) avoiding erroneous execution risks, which can lead to dangerous or even harmful infrastructure failures. The RADON defect prediction tool is well beyond state-of-the-art and current practice. No previous tool offers solutions for infrastructure code automated maintenance (e.g., the focused maintenance facilities allowed by our defect prediction solution) or predictive orchestration error resolution. The *RADON Framework for IaC Defect Prediction* is a first-of-its-kind solution to support developers and operators in infrastructure code maintenance and evolution.

1.1 Deliverable Objectives

The main objective of the deliverable is to document the final version of the DPT architecture and implementation, providing examples of its usage while highlighting its achievements.

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

- A description of the extensions and improvements that led to the final architecture and implementation of the RADON Framework for IaC Defect Prediction.
- A description of its architecture and implementation. In this regard, the deliverable provides a general overview of the framework and its components and a detailed description of their architecture.
- A step-by-step illustration of the DP APIs usage and 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

The main achievements reported in this deliverable reflect that it is possible to:

- crawl open-source Ansible and Tosca repositories from Github and Gitlab;
- mine *up to five types of defects* 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 into the RADON IDE;
- automatically interact with the models through a command-line interface that can be integrated into CI/CD pipelines.

1.3 Structure of the Document

The remainder of this document is organized as follows. Section 2 outlines the framework extensions and improvements since the previous deliverable. Section 3 illustrates the final architecture of the DP and the related tools, while Section 4 shows detailed examples of usage of these tools. Finally, Section 5 concludes the document.

2. Extensions and Improvements

This section covers the main framework modifications to address the KPIs and industrial partners' requirements.

- The framework implementation presented in D3.6 extracted three types of metrics to identify defect-prone scripts: *IaC-oriented, delta,* and *process* metrics. However, experiments performed by Dalla Palma et al. (Dal2021) showed that the models featuring *IaC*-oriented metrics outperform models featuring other metric sets on average. Therefore, for the sake of performance and collection ease, the final implementation uses *IaC*-oriented metrics.

- The framework implementation presented in D3.6 trained defect prediction models using five classifiers, namely *Decision Tree*, *Logistic Regression*, *Naive Bayes*, *Random Forest*, and *SVM* and afterward selected the best model. Dalla Palma et al. (Dal2021) showed that every classifier but *Naive Bayes* reaches high model performance. They also observed that the performance difference between *SVM* and *Logistic Regression* and between *Logistic Regression* and *Decision Tree* is not statistically significant, although the former provided better results most of the time. Consequently, depending on the desired model flexibility and the available computational resources, one can choose them interchangeably without significantly negatively affecting the prediction. Based on these observations, the final implementation relies on the *Decision Tree* classifier as we considered it the best tradeoff between accuracy and interpretability.

- The framework implementation presented in D3.6 predicted an IaC blueprint's defect-proneness as a boolean response, namely *defect-prone* or *defect-free*. In contrast, the current implementation predicts up to five defect types, namely (i) *conditional*, (ii) *configuration data*, (iii) *dependency*, (iv) *service*, and (v) *general*.

- The framework implementation presented in D3.6 supported Ansible only. The current implementation supports TOSCA as well.

The following section describes the final tool architecture in detail.

3. Architecture

This section describes the RADON IaC Defect Prediction's architecture, focusing on its main building-blocks.

3.1 Framework Overview

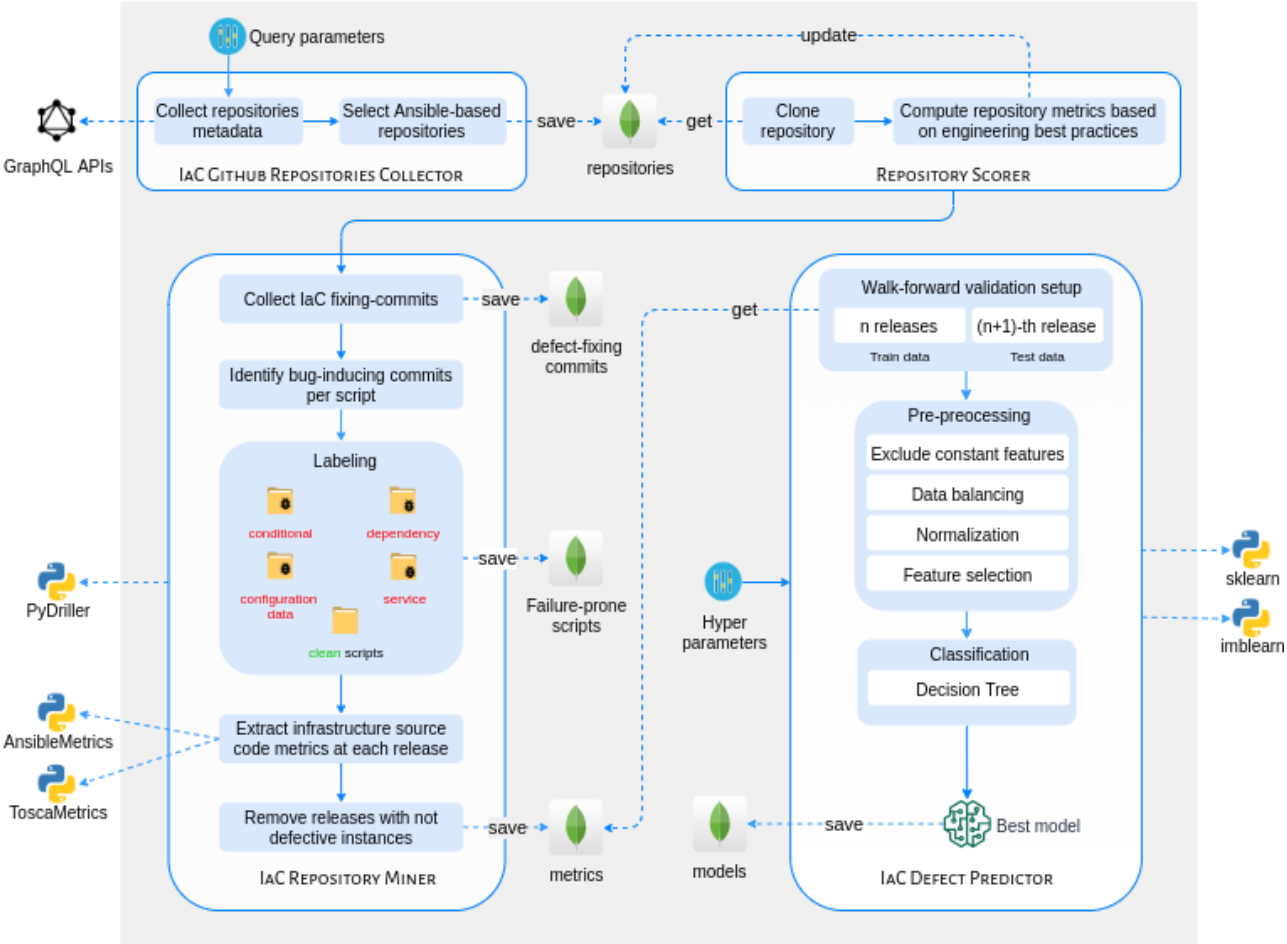

Figure 1 - The RADON Framework for IaC Defect Prediction. The repositories collected by the *Github IaC Repositories Collector* are passed as input to the *Repository Scorer* to pick relevant repositories. Afterward, the *IaC Repository Miner* mines the selected repositories. Its output, consisting of observations of *defect-prone* and *defect-free* IaC scripts for the individual repositories, is used by the *IaC Defect Predictor* to build and evaluate the models.

Figure 1 provides a detailed overview of the proposed framework consisting of four individual components:

- *The Github IaC Repositories Collector* collects active IaC repositories on GitHub.
- *The Repository Scorer* computes repository metrics based on best engineering practices used to select relevant repositories.

- *The IaC Repository Miner* mines failure-prone and neutral IaC scripts from a repository. Then, it gathers a broad set of *IaC-oriented metrics* computed upon the collected IaC scripts to predict their failure-proneness.
- *The IaC Defect Predictor* pre-processes the datasets and trains the Machine Learning models. Given an unseen IaC script, this component classifies it as *failure-prone* or *neutral*.

3.1.1 Github IaC Repositories Collector

The *Github IaC Repositories Collector* searches for public candidate repositories containing Ansible code through the novel GraphQL-based GitHub APIs.[3] The tool is open-source and available on Github[4] and the Python Package Index[5] (PyPI). It enables tuning the GraphQL search query to collect metadata, such as name, description, URL, and root directories, from repositories that match user-defined selection criteria, such as the minimum number of releases, issues, stars, and watchers. Archived, mirrored, and forked repositories are excluded. The metadata are used to select Ansible and Tosca repositories by checking the word *ansible* or *tosca* against their *name* (e.g., *ansible/ansible-examples*), *description* (e.g., *A few starter examples of ansible playbooks*), or, specifically for Ansible, *directory layout* (i.e., the presence of at least two of the following directories: playbooks, meta, tasks, handlers, roles.) The repositories' metadata are then saved on a MongoDB instance and analyzed to mine only relevant projects.

3.1.2 Repository Scorer

Many Github repositories are not for software development, i.e., they are mainly used for experimentation, storage, and academic projects. Therefore, there is a need to identify suitable, well-engineered software projects for pre-trained models that will be made available to the users. In this context, a well-engineered project is a "*software project that leverages sound software engineering practices in one or more of its dimensions such as documentation, testing, and project management*" Munaiah et al. (Mun2017). Furthermore, the models should capture patterns from the most active projects exhibiting the latest configuration management practices. To this end, the RADON Framework for IaC Defect Prediction verifies several constraints, reported in Table 1 along with their description, to select and to mine only relevant projects.

Criterion 1 (i.e., *push events*) allows for discarding inactive projects, while criterion 2 (i.e., *releases*) is needed because the target models are trained at the release-level. The Github IaC Repositories Collector evaluates these criteria. In contrast, the *Repository Scorer*, a Python package

[3] https://developer.github.com/v4
[4] https://github.com/radon-h2020/radon-repositories-collector
[5] https://pypi.org/project/repositories-collector

that we implemented and made available open-source on Github[6] and PyPI[7] evaluates the remaining criteria.

More specifically, the *ratio of IaC Scripts* represents a cutoff to analyze repositories containing IaC scripts that have been determined by the previous works Rahman et al. (Rah2018), Rahman et al. (Rah2019). Criteria 4-9 are considered as useful indicators of well-engineered software projects Munaiah et al. (Mun2017).

The tool takes as input the local path (or remote URL) to a Git repository, calculates metrics related to the last eight criteria reported in Table 1, and saves them in the MongoDB instance along with the repositories' metadata. Once the repository is deemed relevant for the analysis based on the computed metrics, its history is analyzed using the *IaC Repository Miner*.

Please note that, although initially implemented to identify relevant repositories for building pre-trained models, the *Repository Scorer* is also used by the DPT APIs to recommend the best pre-trained model for projects without a sufficient history that would enable model training. Particularly, pre-trained models can be used for such projects based on the similarity between the project's metrics values and the metrics extracted from projects for which pre-trained models exist.

#	Name	Description
1	Push events	The DPT tool should analyze active projects exhibiting the latest practices. Therefore, the repository must have at least one push event to its default branch in the last *n* months. The number of months *n* can be defined by the user, and it is defaulted to *six*
2	Releases	The proposed defect predictor analyzes files at each release and between successive releases. Therefore, the repository must have at least 2 releases
3	Ratio of IaC scripts	Repositories must have sufficient IaC scripts. At least 10% of the files must be IaC scripts
4	Core Contributors	The project must have at least 2 contributors whose total number of commits accounts for 80% or more of the total contributions
5	Continuous Integration	The repository must use a CI service, determined by the presence of a configuration file required by that service (e.g., a .travis.yml for TravisCI)
6	Comment Ratio	The comment ratio must be at least 0.2%
7	Commit Frequency	The commit frequency must be at least 2.0
8	Issue Frequency	The issue frequency must be at least 0.01
9	Lines of Code	The repository must have at least 100 lines of code. It is used to co-assess and control the criteria 4-8

Table 1 - Criteria to select repositories that contain evidence of engineered software projects.

[6] https://github.com/radon-h2020/radon-repository-scorer
[7] https://pypi.org/project/repository-scorer

3.1.3 IaC Repositories Miner

IaC Repository Miner relies on the PyDriller framework [Spad2018] to analyze the project history and extract the *defect-prone* and *defect-free* IaC scripts needed for the analysis.

To this end, first, it identifies defect-fixing commits of IaC files according to the categorization and detection rules proposed by Rahman et al. (Rah2020).

IaC Repository Miner keeps only the commits that modify at least one IaC script. Afterward, it determines their failure-proneness as follows.

```
1:  procedure GETFIXEDFILES(fixingCommits:List[str])
2:      fixedFiles = []
3:
4:      for commit in fixingCommits[NEWEST : OLDEST]
5:          for file in commit.modifiedFiles
6:              if file.type != 'IaC' or file.changeType != 'Change'
7:                  continue
8:
9:              bics = SZZ(commit, file)        ▷ Bug-Inducing Commits
10:
11:             currentFix = FixedFile(file.filepath,
12:                                    fic=commit.sha,
13:                                    bic=bics[OLDEST])
14:
15:             if currentFix not in fixedFiles
16:                 fixedFiles.append(currentFix)
17:             else
18:                 existingFix = fixedFiles.get(currentFix)
19:                 if currentFix.fic is older than existingFix.bic
20:                     fixedFiles.append(currentFix)
21:                 else if currentFix.bic is older than existingFix.bic
22:                     existingFix.bic = currentFix.bic
23:
24:     return fixedFiles
```

Algorithm 2 - Procedure to identify IaC files modified in fixing-commits and their bug-inducing commits.

First, it applies **Algorithm 2** to identify IaC files touched by a defect-fixing commit and their *defect-inducing* commits (i.e., *bug-inducing* commit). We refer to those files as *fixed-files*. It analyzes the commits backward from the most recent to the oldest (line 4). For each *fixed-file* (line 5), it relies on the *SZZ* algorithm (Kim2006) to automatically identify the **oldest** commit that introduced a defect in that script (line 9).[8] Files that have not already been fixed in the previously analyzed commits are added to the list of fixed-files (line 15-16). Otherwise, the following procedure applies:

[8] *IaC Repository Miner* relies on the *SZZ* implemented in PyDriller >= 1.15.

- **Lines 19-20.** If the current fixing commit (i.e., *fic* in **Algorithm 2**) is older than the previously bug-inducing commit of the fixed-file (i.e., *bic* in **Algorithm 2**), then the current fixed-file is added to the list of fixed-files as a new object. This scenario is illustrated in **Figure 2a**. Here, a file has been fixed by a recent commit C10. It fixes a defect introduced in C8 (*bic*), newer than the current defect-fixing commit C4 (*fic*). Consequently, a new `FixedFile(file=A, fic=C4, bic=C1)` is added to the list of *fixed-files*, in addition to the previous `FixedFile(file=A, fic=C10, bic=C8)`.

- **Lines 21-22.** Suppose the current commit is more recent than the bug-inducing commit of the previously fixed-file, and the latter is more recent than the current bug-inducing commit. In that case, the *bic* of the existing fixed-file is updated with the current one. This scenario is illustrated in **Figure 2b**. Here, a previously analyzed commit C8 fixes a defect introduced in C5. So far, the file is considered failure-prone from C5 to C7. Nevertheless, the current commit C6 fixes a defect in the same file introduced in C4. Therefore, the file is considered failure-prone from C4 to C7, and the existing `FixedFile(file=B, fic=C8, bic=C5)` is updated to `FixedFile(file=B, fic=C8, bic=C4)`. This window is expanded if any fix to the same file is found in commits before C5.

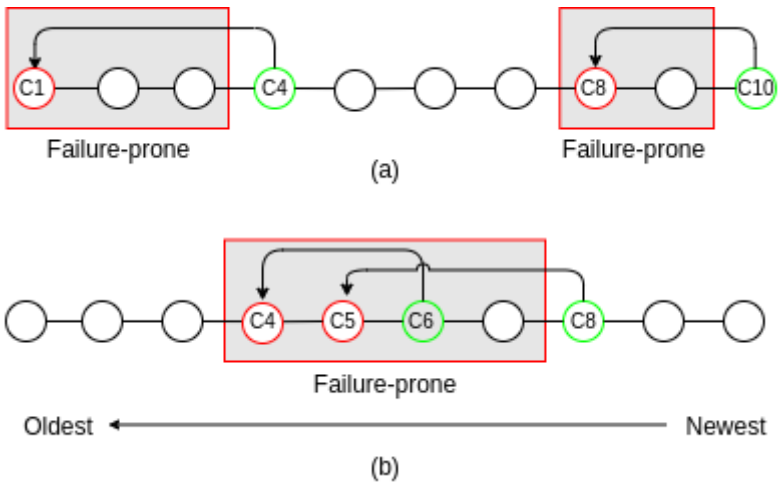

Figure 2 - Two scenarios of the labeling process.

Please note that for the sake of clearness, we omitted the lines handling file-renaming in **Algorithm 2**. Once the procedure ends, and the *fixed-files* list is returned, the labeling process is straightforward: all the snapshots of a fixed-file between its *bic* (inclusive) and the *fic* are labeled with the defect-type fixed by the file's fixing-commits.

Along with the failure-proneness of the IaC scripts, *IaC Repository Miner* gathers a comprehensive set of IaC-oriented (ICO) features to train the defect prediction model derived from previous work

in defect prediction (Dal2020b). More specifically, ICO features, or metrics, are structural properties derived from the IaC source code analysis, which measure and potentially predict the maintainability of IaC scripts. The *IaC Repository Miner uses* the *RADON AnsibleMetrics* (Dal2020a) and *ToscaMetrics* tools to collect 46 and 17 Ansible[9] and TOSCA[10] metrics, respectively.

More particularly, eight of them are "traditional" code metrics adapted for IaC scripts, e.g., the lines of code. These metrics have been long adopted in the context of traditional defect prediction, and, therefore, we selected them to assess their role in predicting IaC defects. They are extracted both from Ansible and from TOSCA blueprints.

Another 14 metrics in this set have already been studied by Rahman et al. (Rah2019) for predicting defects in Puppet code. As such, we considered them to verify their generalizability when employed for predicting Ansible defects.

Finally, the last 24 refer to best and bad practices and data management of Ansible code: in our previous work (Dal2020b), we conjectured that these metrics could negatively affect the maintainability of IaC code and increase its failure-proneness. As an example, let consider the number of distinct modules. IaC scripts consisting of many distinct modules are naturally less self-contained and potentially affect the complexity and maintainability of the system, and, therefore, we considered it among the set of metrics for IaC defect prediction.

Similarly, the remaining 10 Tosca metrics were identified by looking at the TOSCA Simple Profile documentation and are available online.

These metrics are extracted from the IaC scripts at each release, saved in the MongoDB instance for their analysis or use by the *IaC Defect Predictor*.

3.1.4 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, and hyper-parameters. It relies on the *Decision Tree* classifier.

1. **Feature selection**. The data is not always initially intended for defect prediction. Therefore, not all the dataset features may be useful 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 dataset size, speed-up the training, and select the optimal number of features that maximize a given performance criterion.

[9] List of metrics available at https://radon-h2020.github.io/radon-ansible-metrics/apis.html
[10] List of metrics available at https://github.com/radon-h2020/radon-tosca-metrics/tree/master/docs

2. **Data balancing**. Once feature selection is finished, the training data are balanced such that the number of defect-prone instances is equal to the number of defect-free instances for a given class of defects. The RADON framework uses three configurations for balancing, namely (i) no balancing; (ii) random under-sampling of the majority class; and (iii) random over-sampling of the minority class. The imblearn package[11] provides an implementation of both the latter techniques in Python.

3. **Data normalization**. In this step, the training data are normalized by 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]; and (iii) *standardization* of the features by removing the mean and scaling to unit variance.

4. **Classification**. The normalized data and the learning algorithm, i.e., a Decision Tree, 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 *joblib* file containing the best model and a report of its performance across the validation steps. The *joblib* file ensures model persistence so that the model can be used in the future without having to retrain.

[11] https://imbalanced-learn.readthedocs.io/en/stable/api.html

3.2 Architecture View

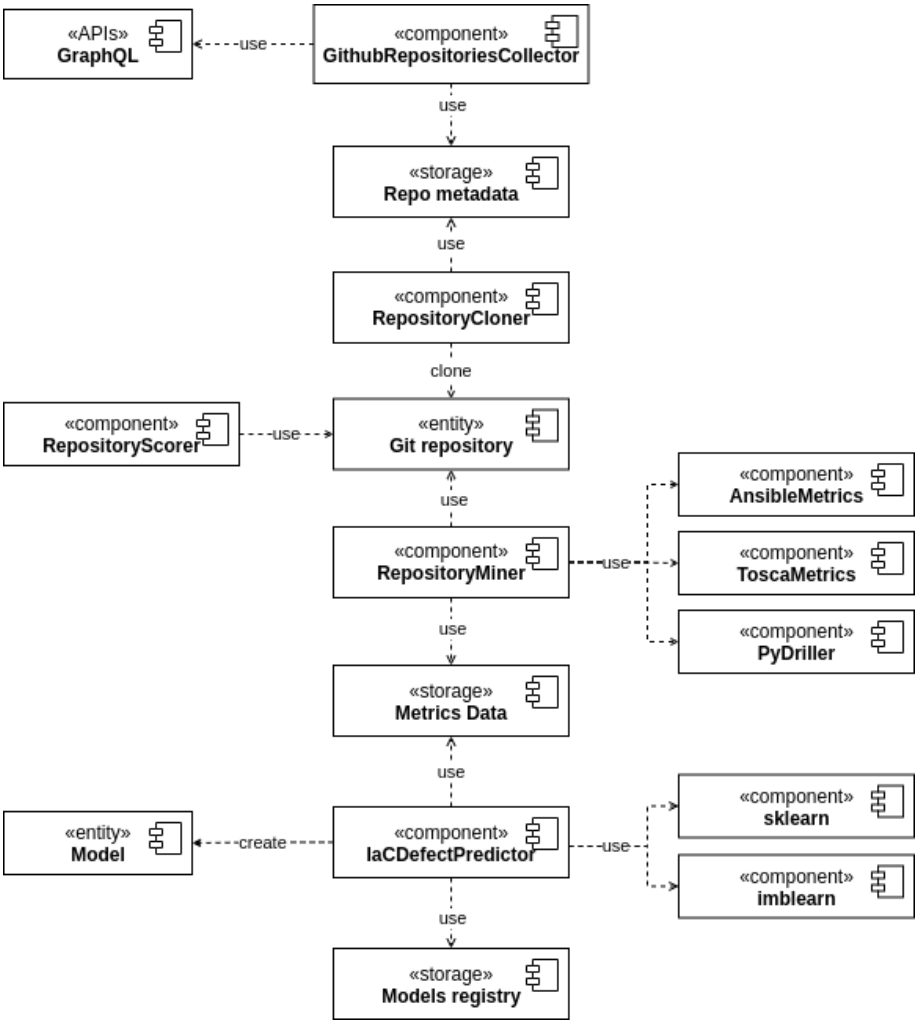

Figure 4 - UML Component diagram of the RADON framework for IaC Defect Prediction.

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

3.2.1 Repositories Crawler

From an architectural perspective, the module *Repositories Crawler* consists of two components (*GithubRepositoryCollector* and *RepositoryScorer*) and relies on a third external component (*GraphQL*). The GithubRepositoryCollector component uses the GraphQL APIs to query Github and collect public repositories created in a given time frame and pushed after a specific date (*date_from*, *date_to*, *pushed_after* of the class diagram depicted in **Figure 5**) through the method `collect_repositories()`. The method `filter_repositories()` is used to filter out repositories based on the user-defined criteria (e.g., minimum number of stars, issues, releases, and watchers.)

In particular, the method `collect_repositories()` generates a json object for each collected repository, containing metadata (such as owner, name, url, main branch, Etc.), and *yields* it to the caller. Here, the *GithubRepositoriesCollector* component envisions two main usage scenario:

1. The caller invokes *GithubRepositoriesCollector*.`collect_repositories()` and waits for all the repositories to be collected before proceeding with the analysis of every repository.
2. The caller invokes *GithubRepositoriesCollector*.`collect_repositories()` and runs the analysis for each repository at a time, as illustrated in **Listing 1** in Section 4.2.1.

The rationale for that is to leave the caller deciding how to employ the tool. The first usage scenario might be faster than the second, but it consumes APIs quotas sooner.

Finally, the repository metadata is saved in the storage, while *RepositoryCollector* clones every repository for the *RepositoryScorer* component to compute the criteria in **Table 1** to decide whether to analyze or discard that repository.

Figure 5 shows the UML class diagram of the *GithubRepositoriesCollector*.

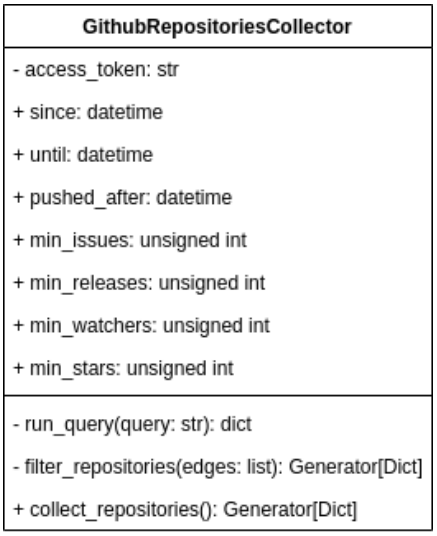

Figure 5 - UML class diagram of *GithubRepositoriesCollector*

3.2.2 IaC Repositories Miner

The *IaCRepositoriesMiner* collects and labels IaC scripts for a given cloned repository by analyzing its commit history using the Python framework *PyDriller*. Then, it runs *AnsibleMetrics*[12] or *ToscaMetrics*[13] on the collected scripts to extract infrastructure source code metrics, depending on the desired language.

[12] https://github.com/radon-h2020/radon-ansible-metrics
[13] https://github.com/radon-h2020/radon-tosca-metrics

It consists of two main modules: `mining` and `metrics`. The former implements the abstract class `BaseMiner` to collect fixing-commits and label failure-prone files based on the defects detectable by the miner. The latter implements the abstract class `BaseMetricsExtractor` to extract source code metrics (*product metrics* in **Figure 7**) from IaC scripts. It is worth knowing that the classes were designed so to ease extensibility to different languages. In particular, `AnsibleMiner` and `ToscaMiner` extend `BaseMiner` to identify fixing-commits and failure-prone files for Ansible and Tosca only, respectively. Similarly, `AnsibleMetricsExtractor` and `ToscaMetricsExtractor` extend `BaseMetricsExtractor` to extract metrics from Ansible and Tosca blueprints, respectively. Using this approach, adding additional IaC languages, such as Chef and Puppet, is straightforward. **Figures 6** and **7** show the UML class diagrams of the *IaC Repository Miner* component.

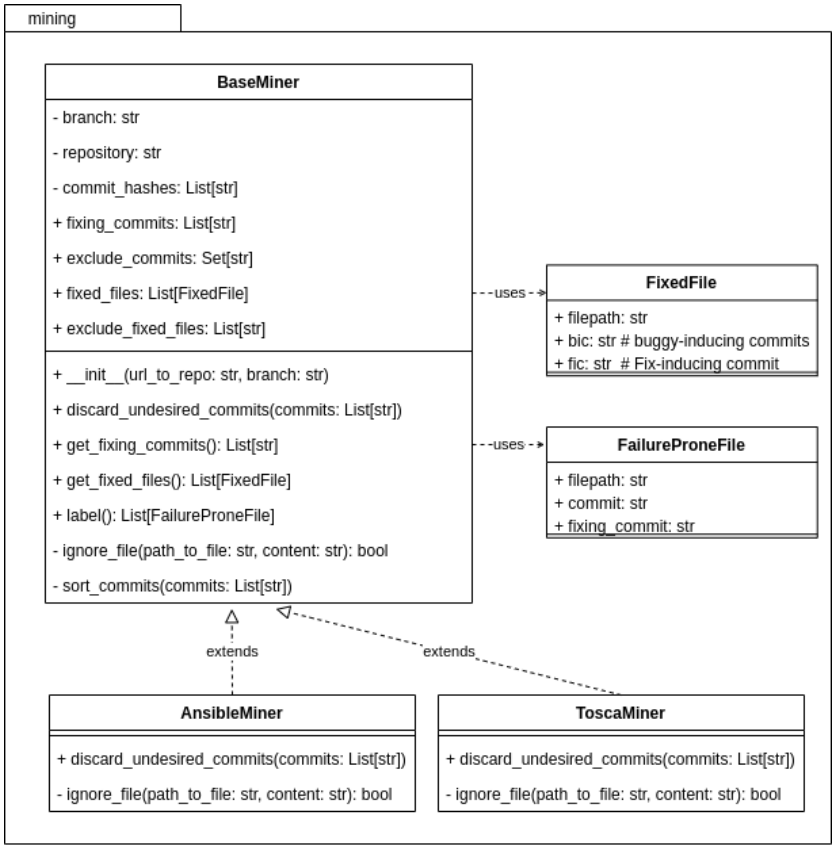

Figure 6 - UML class diagram of the module *mining* of the *IaC Repository Miner* component.

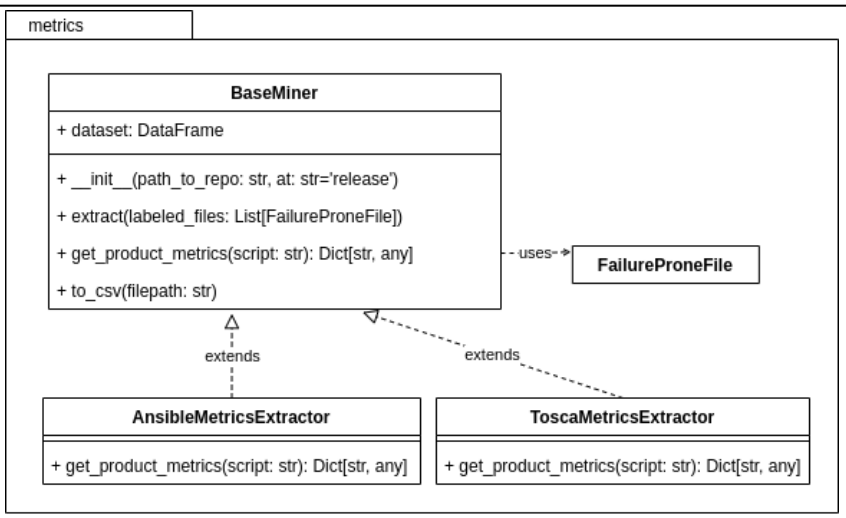

Figure 7 - UML class diagram of the module *metrics* of the *IaC Repository Miner* component.

3.2.3 IaC Defect Predictor

The *IaC Defect Predictor* relies on the Python frameworks *scikit-learn*[14] and *imblearn*[15] 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. It provides a command-line interface (CLI) usable within CI/CD pipelines.

More specifically, the CLI offers three methods: (1) *train* a new model from scratch; (2) *download* a pre-trained model; and (3) *predict new* unseen instances with the model obtained from (1) or (2).

Figure 8 shows the main classes of the IaC Defect Prediction CLI. The IaCDefectPrediction class contains the core attributes and methods to train a new model using different configurations in terms of data balancing, normalization, and classification techniques. The user sets up the configurations through the CLI. Alternatively, a default configuration is used.

After model training, the model is dumped to the hard drive for persistence through the `dump_model()` method. The same model, or the one downloaded, is loaded through the `load_model()` method.

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

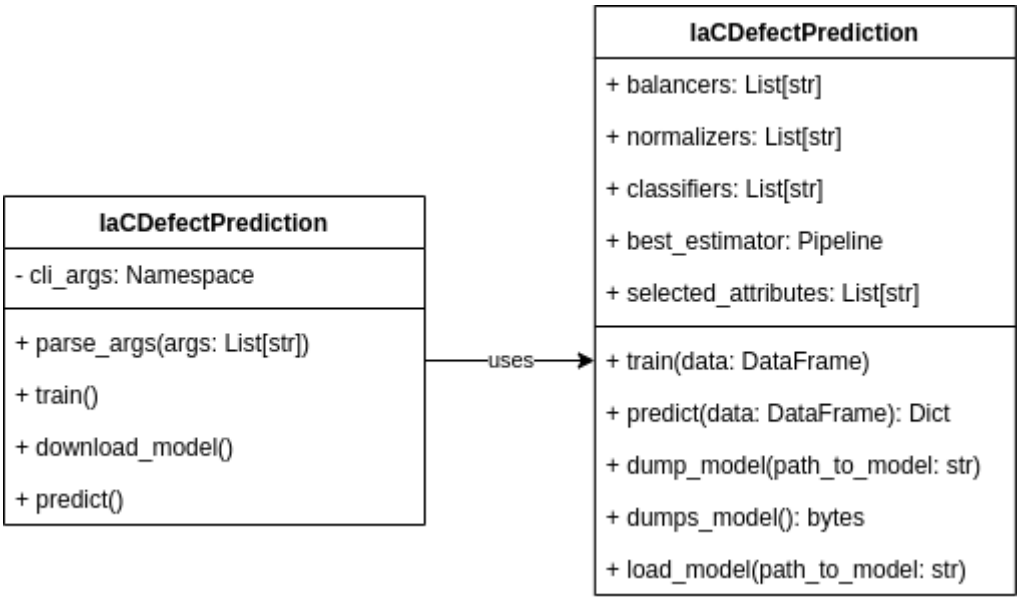

Figure 8 - UML class diagram of the module *metrics* of the *IaC Repository Miner* component.

3.3 Unit Testing

All the test cases for the components above were generated using a test-driven approach.

More specifically, for the *Repository Miner,* first, we documented the Ansible and TOSCA metrics with the intended behavior and examples. Then, we implemented the test cases before the production code to improve our confidence in the metric miners.

The *IaC Defect Predictor* component uses the metrics miner. Therefore, at that point, we were sure the miner behaved as intended.

Finally, we generated tests for each of the DPT commands (i.e., train, download model, and predict) to ensure that the tool provided the relevant and expected outputs. For example, check that a model is saved on the machine after "training" or "downloading" it.

4. Demonstration of use

This section describes how to use the components of the Defect Prediction Tool.

4.1 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 specific contexts, but they aim to explain and show the general usage of the tool. In particular, how to (i) search for relevant IaC-based repositories on GitHub; (ii) mine repositories to collect data (i.e., metadata and metrics for defect-prone and defect-free scripts); (iii) pass data to a Machine Learning classifier for its training and validation.

4.2 GitHub IaC Repositories Collector

This step is needed to collect relevant candidate repositories to train models that the community can use. In contrast, if the user wants to train a specific project model, go to Section 5.3.

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

```
pip install repositories-collector
```

The tool is ready to use!

4.2.1 APIs Usage

The following code snippet shows an example of usage of the `GithubRepositoriesCollector` APIs to crawl GitHub and filter repositories based on several criteria like the date of creation, minimum number of releases, Etc.

Please, note that the tool uses Github's GraphQL APIs. Therefore, the user must create their token to crawl repositories. See how to get one at https://github.com/settings/tokens.

Afterward, either the user exports that token as an environment variable (recommended) with:

```
export GITHUB_ACCESS_TOKEN=***token***
```

on Linux, or they can pass it directly to the Python API.

```
import os
from datetime import datetime
from repocollector import GithubRepositoriesCollector

github_crawler = GithubRepositoriesCollector(
     access_token=os.getenv('GITHUB_ACCESS_TOKEN'),  # or paste your token
     since=datetime(2019, 12, 31),
     until=datetime(2020, 12, 31),
     pushed_after=datetime(2020, 6, 1),
     min_issues=0,
     min_releases=2,
     min_stars=0,
     min_watchers=0)

for repo in github_crawler.collect_repositories():
  print(repo)

  # clone repo, or
  # discard repo, or
  # do something else
```

Listing 1 - Python APIs to collect repositories from Github

4.2.2 Command-line Usage

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

```
usage: repositories-collector [-h] [-v] [--from DATE_FROM]
                                    [--to DATE_TO] [--pushed-after DATE_PUSH]
                                    [--min-issues MIN_ISSUES]
                                    [--min-releases MIN_RELEASES]
                                    [--min-stars MIN_STARS]
                                    [--min-watchers MIN_WATCHERS] [--verbose]
                                    dest

A Python library to collect repositories metadata from GitHub.

positional arguments:
  dest                  destination folder for report

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  --from DATE_FROM      collect repositories created since this date (default:
                        2014-01-01 00:00:00)
  --to DATE_TO          collect repositories created up to this date (default:
                        2014-01-01 00:00:00)
  --pushed-after DATE_PUSH
                        collect only repositories pushed after this date
                        (default: 2019-01-01 00:00:00)
  --min-issues MIN_ISSUES
```

```
                      collect repositories with at least <min-issues> issues
                      (default: 0)
  --min-releases MIN_RELEASES
                      collect repositories with at least <min-releases>
                      releases (default: 0)
  --min-stars MIN_STARS
                      collect repositories with at least <min-stars> stars
                      (default: 0)
  --min-watchers MIN_WATCHERS
                      collect repositories with at least <min-watchers>
                      watchers (default: 0)
  --verbose           show log (default: False)
```

Listing 2 - Command-line usage of repositories-collector

Please note that it is mandatory to export the token in the environment variable GITHUB_ACCESS_TOKEN.

4.3 IaC Repositories Miner

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

```
pip install repository-miner
```

The tool is ready to use!

4.3.1 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.

```
from repominer.mining.ansible import AnsibleMiner
from repominer.metrics.ansible import AnsibleMetricsExtractor

miner = AnsibleMiner('https://github.com/owner/repository')
miner.get_fixing_commits()
miner.get_fixing_files()
failure_prone_files = miner.label()

metrics_extractor = AnsibleMetricsExtractor()
metrics_extractor.extract(failure_prone_files)
  print(metrics_extractor.dataset.head())
```

Listing 3 - Python APIs to mine a Git repository on Github

Listing 3 instantiates an `AnsibleMiner` object to mine failure-prone files that are passed to the `AnsibleMetricsExtractor` to extract Ansible source code metrics from those files. The users can access the metrics through the attribute `AnsibleMetricsExtractor.dataset`.

Although Listing 3 targets Ansible, TOSCA repositories can be mined as well. To do so, replace every occurrence of `Ansible` with `Tosca` and `ansible` with `tosca`.

4.3.2 Command-line Usage

This section exemplifies the command-line usage through two primary usage scenarios: mine *failure-prone files* and *extract metrics*.[16]

4.3.2.1 Failure-prone Files Mining

The `mine failure-prone-files` command allows for mining defect-prone files that have been fixed by fixing-commits, as described in Section 3.1.3. Below is an example of usage:

```
repo-miner mine failure-prone-files github ansible adriagalin/ansible.motd
                        path/to/results/ --verbose
```

A file `failure_prone_files.json` is saved in `path/to/results/` containing a list of JSON objects representing defect-prone files, with the information present in the `FailureProneFile` class shown in **Figure 6**. The command's output is illustrated below:

```
Mining adriagalin/ansible.motd [started at: 14:28]
Identifying fixing-commits
Saving 4 fixing-commits [14:28]
JSON created at ./fixing-commits.json
Identifying ansible files modified in fixing-commits
Saving 3 fixed-files [14:28]
JSON created at ./fixed-files.json
Identifying and labeling failure-prone files
Saving failure-prone files
JSON created at ./failure-prone-files.json
```

[16] The cli documentation can be found online at https://radon-h2020.github.io/radon-repository-miner/cli.mining.html

where the file containing information about the fixing-commits and failure-prone files are in bold.

Figure 9 and **Figure 10** show an extract of `fixing-commits.json` and `failure-prone-files.json`, respectively.


```
[{
     "filepath": "tasks/main.yml",
     "fic": "e283a1f673b1bd583f2a40645671179e46c9048f",
     "bic": "033cd106f8c3f552d98438bf06cb38e7b8f4fbfd"
 },{...}]
```


Figure 9 - Example of content for `fixing-commits.json`


```
[{
     "filepath": "tasks/main.yml",
     "commit": "3a8d9b7ed430a3367d8d8616e0ba5d2bddb07b9e",
     "fixing_commit": "e283a1f673b1bd583f2a40645671179e46c9048f"
 },{
     "filepath": "tasks/main.yml",
     "commit": "5190e2a0325bd3e7f9112dbf5a7f67cfa19fb3ae",
     "fixing_commit": "e283a1f673b1bd583f2a40645671179e46c9048f"
 },{
     "filepath": "meta/main.yml",
     "commit": "84c7e12d2510db7e0ee20cc343c0e1676de41bc2",
     "fixing_commit": "f9ac8bbc68dedb742e5825c5cf47bca8e6f71703"
 }]
```


Figure 10 - Example of content for `failure-prone-files.json`

4.3.2.2 Metrics Extraction

The `failure_prone_files.json` is used to extract metrics from each of the files it contains through the command `extract-metrics`. Below is an example of how to run it:

```
repo-miner extract-metrics https://github.com/adriagalin/ansible.motd
path/to/failure-prone-files.json ansible product release path/to/results/
--verbose
```

The command above specifies that only Ansible-related (`ansible`) product metrics (`product`) must be extracted at each project release (`release`). A file `path/to/results/metrics.csv` is created containing the metrics values extracted from the observations (i.e., files) of each project across the project history (i.e., commits). It looks like **Figure 11.**

avg_task_size	commit	failure_prone	filepath
2.0	4c127cd1b9da3ce99de1fe9e6ad05b04fa3b8007	0	demo/molecule/tasks/debian.yml
14.0	4c127cd1b9da3ce99de1fe9e6ad05b04fa3b8007	0	demo/molecule/meta/main.yml
2.0	4c127cd1b9da3ce99de1fe9e6ad05b04fa3b8007	0	demo/molecule/tasks/rhel.yml
3.0	4c127cd1b9da3ce99de1fe9e6ad05b04fa3b8007	1	demo/molecule/tasks/main.yml
2.0	72a669aab8eff892bf6c6338b06ec526d174f112	0	demo/molecule/tasks/debian.yml
14.0	72a669aab8eff892bf6c6338b06ec526d174f112	0	demo/molecule/meta/main.yml
2.0	72a669aab8eff892bf6c6338b06ec526d174f112	0	demo/molecule/tasks/rhel.yml
3.0	72a669aab8eff892bf6c6338b06ec526d174f112	1	demo/molecule/tasks/main.yml
2.0	e87738d31c132d42aeec8b605104b8f42962a567	0	demo/molecule/tasks/debian.yml
14.0	e87738d31c132d42aeec8b605104b8f42962a567	0	demo/molecule/meta/main.yml
2.0	e87738d31c132d42aeec8b605104b8f42962a567	0	demo/molecule/tasks/rhel.yml
3.0	e87738d31c132d42aeec8b605104b8f42962a567	1	demo/molecule/tasks/main.yml
8.0	8083a3897dbfac82213fa22c21fe7665a696378b	0	molecule/cookiecutter/galaxy_init/{{co
	8083a3897dbfac82213fa22c21fe7665a696378b	0	molecule/cookiecutter/galaxy_init/{{co

Figure 11 - Example of content for `metrics.csv`

4.3.3 Docker Usage

For the sake of portability, the *Repository Miner* comes with an additional Docker image available on DockerHub.[17]

First, pull the image:

```
docker pull radonconsortium/repo-miner:latest
```

Then, create a folder to use as a host volume to share data and results between the host machine and the Docker container. For example (on Linux):

```
mkdir /tmp/repo-miner-volume
```

Finally, run the *mine* and *extract metrics* command as follows:

```
docker run -v /tmp/repo-miner-volume:/app radonconsortium/repo-miner repo-miner
  mine failure-prone-files github ansible adriagalin/ansible.motd . --verbose
```

[17] https://hub.docker.com/r/radonconsortium/repo-miner

```
docker run -v /tmp/repo-miner-volume:/app radonconsortium/repo-miner repo-miner
        extract-metrics https://github.com/adriagalin/ansible.motd
    path/to/failure-prone-files.json ansible product release . --verbose
```

Please note that the user's volume `/tmp/repo-miner-volume` is now associated to the container's working directory `/app` through the command `/tmp/repo-miner-volume:/app`. In this way, every file stored in `/tmp/repo-miner-volume` is visible by the container in `/app`, and any new container's created file will be visible by the user in the volume.

Also note that the path `path/to/results/` MUST be replaced by the relative path `.` (dot). In this way, the results are saved within the container's working directory **`/app`**, and can be accessed on the user's machine in **`/tmp/repo-miner-volume`** (or the user's defined volume).

4.4 IaC Defect Predictor

The IaC Defect Predictor consists of two tools: one implementing the endpoints to download and use pre-trained models (https://github.com/radon-h2020/radon-defect-prediction-endpoints); and a client, enabling model training, pre-trained model download, and model usage for predictions (https://github.com/radon-h2020/radon-defect-prediction-cli).

First, download and install the client from PyPI (https://pypi.org/project/radon-defect-predictor/):

```
pip install repository-miner
```

The tool is ready to use!

4.4.1 Python APIs Usage

Listing 4 trains a new classifier from scratch on a given dataset (*train_dataset*), using a *minmax* normalization to scale features between 0 and 1, and a Decision Tree (*dt*) classifier. Finally, it uses the trained defect prediction model to predict a new observation (*test_observation*).

Please note, this example requires that the user generates a training set using the Repository Miner.

```
import pandas as pd
from radondp.predictors import DefectPredictor

def usage_example(train_dataset: pd.DataFrame, test_observation: pd.DataFrame):
    dp = DefectPredictor()
    db.normalizers(['minmax'])
    dp.classifiers(['dt'])
    dp.train(train_dataset)

    # Dump estimator, selected features, and model's validation report to disk
```

```
    dp.dump_model('path/to/models/') to persist the model

    print(dp.predict(test_observation))

# TODO Create train_dataset, test_dataset
usage_example(train, test)
```

Listing 4 - Python APIs to train a Defect Prediction model

4.4.2 Command-line Usage

The command-line tool was designed to ease the use of the DPT within the CI/CD pipeline. This section exemplifies the command-line usage through three main usage scenarios: training *a new model, downloading a pre-trained model,* and *predicting instances*.[18]

4.4.2.1 Model Training

Below you can find the CLI usage for training a model from scratch.

```
usage: radon-defect-predictor train [-h] [--balancers BALANCERS] [--normalizers
NORMALIZERS] path_to_csv classifiers

positional arguments:
  path_to_csv       the path to the csv file containing the data for training
  classifiers       a list of classifiers to train, i.e., dt, logit,  nb, rf, svm

optional arguments:
  -h, --help        show this help message and exit
  --balancers BALANCERS
                    the balancers to balance training data, i.e., none, rus, ros
  --normalizers NORMALIZERS
                    the normalizers to normalize data, i.e., none, minmax, std
```

Listing 5 - CLI usage to train a Defect Prediction model

Assuming the user wants to train a new Decision Tree-based defect prediction model for the repository *ansible-community/molecule.*[19] Provided that (s)he has already generated a training dataset *molecule.csv*[20] using the radon-miner and it is present in the current working directory, she can train the model as follows:

[18] The cli docs is available at https://radon-h2020.github.io/radon-defect-prediction-cli/cli/getting_started/
[19] https://github.com/ansible-community/molecule
[20] Downloadable at https://radon-h2020.github.io/radon-defect-prediction-cli/examples_resources/molecule.csv

```
radon-defect-predictor train ./molecule.csv dt
```

The model is persisted in the user's working directory as *radondp_model.joblib* to predict new instances, as shown later in Section 4.4.2.3.

4.4.2.2 Model Download

Alternatively, the user might want to download a pre-trained model for several reasons, mainly because the user's project does not have a sufficient defect history to enable the training within the same project. It is worth noting that the pre-trained model is taken from the project that most resembles the user's project, based on the metrics extracted from *Repository Scorer*. In this case, the metrics extraction is handled by the *IaC Defect Predictor*.

```
usage: radon-defect-predictor download-model [-h] {ansible,tosca} {github,gitlab}
repository

positional arguments:
  {ansible,tosca}  the language the model is trained on
  {github,gitlab}  the platform the user's repository is hosted to
  repository       the user's remote repository (<namespace>/<repository>)

optional arguments:
  -h, --help       show this help message and exit
```

Listing 6 - CLI usage to download a Defect Prediction model

The model is saved in the user's working directory as *radondp_model.joblib*.

4.4.2.3 Instance Prediction

Once the user has trained a model or downloaded a pre-trained model, they can use it for predicting the failure-proneness of a new observation (i.e., metrics extracted from an IaC script).

```
usage: radon-defect-predictor predict [-h] language path_to_artefact

positional arguments:
  {ansible,tosca}   the language of the file
  path_to_artefact  the path to an Ansible file or Tosca file or .csar

optional arguments:
```

```
    -h, --help            show this help message and exit
```

Listing 6 - CLI usage to predict an IaC blueprint using a Defect Prediction model

Provided that an Ansible model *radondp_model.joblib* is present in the user's working directory and that the file to analyze is located in */tmp/ansible.yml,* then the following command

```
radon-defect-predictor predict ansible /tmp/ansible.yml
```

generates a new file called *prediction_results.json* in the user's working directory. The *json* reports the following information:

- *file*: the file path of the analyzed file;
- *failure_prone:* whether the file is predicted failure-prone;
- *analyzed_at:* to log the analysis timestamp.

Please note, as new runs are executed within the same working directory, the new predictions are **appended** to *prediction_results.json.*

4.4.3 Docker Usage

For the sake of portability, the *IaC Defect Predictor* comes with an additional Docker image available on DockerHub.[21]

First, pull the image:

```
docker pull radonconsortium/radon-dp:latest
```

Then, create a folder to use as a host volume to share data and results between the host machine and the Docker container. For example (on Linux):

```
mkdir /tmp/radon-dp-volume
```

Finally, provided an Ansible file is present in the volume `/tmp/radon-dp-volume (e.g., ansible.yml)` run it as follows:

```
docker run -v /tmp/radon-dp-volume:/app radonconsortium/radon-dp predict
                        ansible ./ansible.yml
```

See Section 4.4.2 on how to use the command-line options. The model and the results can be found in the volume folder */tmp/radon-dp-volume.*

[21] https://hub.docker.com/r/radonconsortium/radon-dp

4.4.4 RESTFul APIs Usage

The RADON Defect Prediction RESTful APIs provide two primary endpoints:

- `/models` to collect a pre-trained model suitable for the project at hand;
- `/predictions` to predict a new instance given a pre-trained model.

4.4.4.1 Getting a Pre-trained Model

The `/models` endpoint accepts a GET request containing the following information as parameters:

- the `language` the model was trained on, i.e., Ansible or TOSCA;
- the `comments_ratio`, i.e., the number of commented lines on the total lines in the project;
- the `commit_frequency`, i.e., the average number of project's commits per month;
- the `core_contributors`, i.e., the minimum number of contributors that contributed for at least 80% of the commits in the project.
- whether the project has_ci, i.e., it uses continuous integration;
- whether the project has_license, i.e., it specifies a LICENSE file;
- the `iac_ratio`, i.e., the number of IaC files on the total number of files in the project;
- the `issue_frequency`, i.e., the average number of project's issue events per month;
- the `repository_size`, i.e., the total number of the project's executable source lines of code.

Please note, the parameters above mainly refer to those listed in **Table 1**.

To test the command, type the following URL in your Browser's search bar:

https://radon-defect-prediction.herokuapp.com/models?language=ansible&has_license=1&iac_ratio=0.8

The response is a JSON object containing the following information:

- the `model_id`, i.e., the id of the most similar project for which a pre-trained model exists. The id must be passed to the `/predictions` to use the pre-trained model for predicting a new instance;
- a `similarity score` that ranges in [0, 1] (the higher, the better), and that indicates how similar are the user's and model's project;
- a list of information about the pre-trained `models` available for that project. Specifically, a set of rules describes how a model classifies files as defect-prone or not
- the defect `type` that that model can identify.

For example, the above URL call returns the following response, which shows two pre-trained models (i.e., one predicting *configuration data*-related defects and a *general* model). Besides, the

similarity between the models' project and the user's project is only 0.006. Passing the remaining parameters (i.e., more information) to the API call can eventually identify more suitable models.


```
{"model_id":78658768,
 "models":[
  {"rules":"|--- num_file_modules <= 0.80\n|
                |--- class: 0.0\n
           |--- num_file_modules >  0.80\n
                |--- class: 1.0\n",
    "type":"configuration_data"},
   {"rules":"...", "type":"general"}],
 "similarity":0.0060594480384355
}
```


However, for the sake of functionality, the user only needs to know the model's id.

4.4.4.2 Using a Pre-trained Model for Predictions

The `/predictions` endpoint accepts a GET request containing the following information as parameters:

- the `model_id` returned by the call to `/models`;
- the `language` of the script to analyze, i.e., Ansible or TOSCA;
- a list of Ansible or Tosca metrics extracted from the blueprint, e.g., `num_tasks` for Ansible and `num_node_types` for Tosca.

To test the command, type the following URL in your Browser's search bar:

https://radon-defect-prediction.herokuapp.com/predictions?language=ansible&model_id=78658768&num_names_with_vars=10&num_ignore_errors=3&num_conditions=1

The response is a JSON object containing the following information:

- whether the file is `failure_prone` based on the metrics passed as parameters;
- a list of defect `types`, i.e., `general`, `conditional`, `configuration data`, `dependency`, or `service`, with
- a list of `decisions` for each defect type consisting of tuples (metric, comparison sign, value) indicating why the model predicted a file as failure-prone for a given defect.

For example, the above URL call returns the following response, which denotes that the file is failure-prone, without specifying the type (i.e., `general`). It also gives a possible cause of the defect, that is, `avg_task_size=0` and `num_conditions>0.5`.

```
{
 "defects":[{
     "decision":[["avg_task_size","<=",0.0],["num_conditions",">",0.5]],
     "type":"general"
    }],
 "failure_prone":true
}
```


It is worth noting that the user does not have to pass these parameters manually. Rather, they can use the `radon-defect-prediction-cli` or `radon-defect-prediction-plugin`.

5. Conclusions

This deliverable presented the final version of the Defect Prediction tool by revisiting DP's initial version, giving a detailed overview of the extended DP's architecture and implementation, as well as a demonstration of usage.

The final version of the DP supports:

- crawling of open-source repositories from Github and Gitlab;
- mining of defect-prone and defect-free instances from the collected repositories to build classifiers to detect defective *Ansible and Tosca blueprints* as described in KPI1.
- training of those classifiers to identify *five different defect types*, as described in KPI2, and their prediction usage.

The defect prediction tool can be used as (i) a standalone portable plugin for Visual Studio Code and Eclipse Che, allowing users to use the models graphically. The plugin is integrated within the RADON IDE, as described in [D2.7] RADON Integrated Framework I; (ii) a command-line client usable within CI/CD pipeline. Finally, as future works, we plan to extend the tool by

- Supporting more IaC languages and technologies. For example, starting from the most popular such as Docker, Kubernetes, Chef, and Puppet.

- Supporting more defect types. Current defect types are based on the defect taxonomy defined by Rahman et al. [Rah2020]. As soon as new defects are elicited, we plan to integrate them into the presented tool.

- Supporting more fine-grained defect prediction models. Current models are trained at the file-level, meaning that they cannot indicate the exact location (i.e., module or source code line) where the bugs are expected to occur. Future works will focus on identifying defects and training models on a module- or line-level. These models will instrument the user with even more helpful clues about the presence and location of defects in the IaC source code, albeit it is more labor-intense and challenging to collect relevant data.

Appendix

A.1 Compliance with Requirements

The following table lists the DP-related requirements as published in the deliverable [D3.6]. However, lessons learned from the evaluation of the previous version of the defect prediction tool led to N changes in the requirements for the defect prediction tool's final version:

- Updated R-T.3.4-8, which states that the defect prediction tool (mainly, the integrated plugin) must provide filters to decide which predefined defects to find. We updated this requirement to analyze all different types of defects simultaneously and report only the found ones. Then, a list of predicted defects is reported to the user through the plugin. So, the user does not have to specify which defects to seek. This aspect improves the overall usability of the tool.

- Replaced R-T.3.4-10, which states that the defect prediction tool must improve performances over manual inspection. Unfortunately, the evaluation of this requirement is impractical, as it would require a controlled experiment or an ethnography. Such study should involve multiple developers or operators detecting manually injected faults using or not the defect prediction tool; at this stage and under the COVID pandemic, such an originally planned controlled experiment cannot be performed. Conversely, considering that the goal of the tool is to prioritize manual inspection to improve the overall code quality, we planned an industrial survey to be performed with RADON industrial case-study owners and External Advisory Board (EAB) partners to understand whether they perceive the defect prediction tool helpful in their daily tasks. The results will be published in the RADON deliverable D6.5 *Final assessment report*. A draft of the survey is available in Section A.3 of the appendix.

- Canceled R-T.3.4-7, which states that the defect prediction tool must provide a linter to flag programming errors, bugs, style, and warnings. The defect prediction already provides information on whether an IaC script contains bugs. Although, it is worth noting that the defect prediction models are trained at the file-level. Therefore, it cannot indicate the location (i.e., source code line) where the bugs are expected to occur. However, the defect prediction tool provides rules to interpret the prediction (i.e., the metrics and respective values that led to predicting a file as failure-prone). Finally, a linter for Ansible[22] is already available. Another for TOSCA is in development[23].

[22] https://ansible-lint.readthedocs.io/en/latest/
[23] https://github.com/tliron/puccini-language-server

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

- ✔ (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.

Please note, a green check (✔) indicates how the new level of compliance *improved* since deliverable [D3.6].

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	**Cancelled (See A.1)**
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	**Cancelled (See A.1)** **Although DPT works at the file level, it already provides rules to interpret the prediction. Furthermore, A linter for Ansible is already available[24] and another for TOSCA is in development[25].**

[24] https://ansible-lint.readthedocs.io/en/latest/
[25] https://github.com/tliron/puccini-language-server

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	**Replaced (See A.1)** **For this requirement, we needed a controlled experiment or an ethnography, which is not practical due to COVID circumstances.** **As a partial replacement, we planned a survey with RADON industrial case-study owners and EAB partners to understand whether DPT is helpful in their daily tasks. The results will be published in D6.5 Final assessment report.**
* This requirement was canceled. See Section A.1 for more info ** This requirement was updated. See Section A.1 for more info			

A.2 Key Performance Indicators

We initially defined two key performance indicators (KPIs) for the Defect Prediction Tool, relevant for this deliverable as they impose requirements on functionality:

ID	Description	Level of compliance
KPI 1	At least 2 IaC Languages Covered	✔✔✔
KPI 2	Up to 5 IaC Defect Types	✔✔✔

A.3 Defect Prediction Tool Validation Survey

In the following, we provide the survey we defined to evaluate the Defect Prediction Tool from a qualitative perspective.

Validation context: ENG-Ansible/ENG-TOSCA/PRQ Survey number: NUMBER Did you use the plugin or the command-line tool via Jenkins? PLUGIN/JENKINS
Does the DPT report any problematic metric according to their distribution? YES/NO If so, answer the following for each of them:

What is the name of the metric? METRIC NAME On a scale of 1 to 5, how do you rate the severity of the reported metric value? (1-5) NOTE: by severity, we denote the impact that the metric score reflects on your refactoring or maintenance activities on the infrastructure code. (1 represents it being not severe, and 5 representing it to be very severe): Could you explain why you consider it severe or not? OPEN QUESTION
Does the DPT report any defects? YES/NO If so, answer the following for each of them: What defect is exhibited? DEFECT NAME (i.e., failure-prone, conditional, configuration data, dependency, service) On a scale of 1 to 5, how do you rate the severity of the defect? (1-5) (1 represents it being not severe, and 5 representing it to be very severe) Could you explain why you consider it severe or not? OPEN QUESTION On a scale of 1 to 5, is the explanation of the defect clear? (1-5) (1 represents it being not clear, and 5 representing it to be very clear) Could you please explain why? OPEN QUESTION
On a scale of 1 to 5, to what extent do you agree with the following statements? (1 represents strongly disagree, and 5 represents strongly agree) The use of DPT could help me to improve IaC quality. (1-5) I think that I could easily learn how to use DPT. (1-5) I think it is a good idea to use DPT to improve IaC quality. (1-5) The use of DPT may imply major changes in IaC development. (1-5) I feel comfortable using a tool to monitor IaC quality. (1-5) I have the intention to use DPT in my company when it becomes available. (1-5)
Would you suggest any additional defects? If so, could you please name it and explain why? OPEN QUESTION

References

[D2.7] RADON Integrated Framework I

[D3.6] RADON Defect Prediction Tool I

[Dal2020a] Dalla Palma S, Di Nucci D, Tamburri DA. AnsibleMetrics: A Python library for measuring Infrastructure-as-Code blueprints in Ansible. SoftwareX. 2020 Jul 1;12:100633.

[Dal2020b] Dalla Palma S, Di Nucci D, Palomba F, Tamburri DA. Towards a catalogue of software quality metrics for infrastructure code. Journal of Systems and Software. 2020 Jul 8:110726.

[Dal2021] Dalla Palma S, Di Nucci D, Palomba F, Tamburri DA. Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics. IEEE Transactions on Software Engineering. 2021 Jan 13.

[Gue2019] Guerriero M, Garriga M, Tamburri DA, Palomba F. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In2019 IEEE International Conference on Software Maintenance and Evolution (ICSME) 2019 Sep 1 (pp. 580-589). IEEE.

[Kim2006] Kim S, Zimmermann T, Pan K, James Jr E. Automatic identification of bug-introducing changes. In21st IEEE/ACM international conference on automated software engineering (ASE'06) 2006 Sep 18 (pp. 81-90). IEEE.

[Mun2017] Munaiah N, Kroh S, Cabrey C, Nagappan M. Curating github for engineered software projects. Empirical Software Engineering. 2017 Dec;22(6):3219-53.

[Rah2018] Rahman A, Williams L. Characterizing defective configuration scripts used for continuous deployment. In2018 IEEE 11th International conference on software testing, verification and validation (ICST) 2018 Apr 9 (pp. 34-45). IEEE.

[Rah2019] Rahman A, Williams L. Source code properties of defective infrastructure as code scripts. Information and Software Technology. 2019 Aug 1;112:148-63.

[Rah2020] Rahman A, Farhana E, Parnin C, Williams L. Gang of eight: a defect taxonomy for infrastructure as code scripts. In2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE) 2020 Oct 5 (pp. 752-764). IEEE.

[Spa2018] Spadini D, Aniche M, Bacchelli A. Pydriller: Python framework for mining software repositories. InProceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering 2018 Oct 26 (pp. 908-911).