SoftwareX 12 (2020) 100633

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

AnsibleMetrics: A Python library for measuring Infrastructure-as-Code blueprints in Ansible

Stefano Dalla Palma [a,*], Dario Di Nucci [a], Damian A. Tamburri [b]

[a] Tilburg University - Jheronimus Academy of Data Science, 's-Hertogenbosch, The Netherlands
[b] Eindhoven University of Technology - Jheronimus Academy of Data Science, 's-Hertogenbosch, The Netherlands

ARTICLE INFO

Article history:
Received 29 May 2020
Received in revised form 16 September 2020
Accepted 17 November 2020

Keywords:
Infrastructure as Code
Software metrics
Software quality

ABSTRACT

Infrastructure-as-Code (IaC) has recently received increasing attention in the research community, mainly due to the paradigm shift it brings in software design, development, and operations management. However, while IaC represents an ever-increasing and widely adopted practice, concerns arise about the need for instruments that help DevOps engineers efficiently maintain, speedily evolve, and continuously improve Infrastructure-as-Code. In this paper, we present AnsibleMetrics, a Python-based static source code measurement tool to characterize Infrastructure-as-Code. Although we focus on Ansible, the most used language for IaC, our tool could be easily extended to support additional formats. AnsibleMetrics represents a step forward towards software quality support for DevOps engineers developing and maintaining infrastructure code.

Code metadata

Current code version	0.3.8
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2020_231
Legal Code License	*Apache License, 2.0 (Apache-2.0)*
Code versioning system used	*Git*
Software code languages, tools, and services used	*Python* >= 3.6; *Visual Studio Code* >= 1.48
Compilation requirements, operating environments	*Linux, Windows*
Developer documentation/manual	https://radon-h2020.github.io/radon-ansible-metrics

1. Motivation and significance

DevOps is a family of tools and techniques that shorten the software lifecycle and intermix software development activities with IT operations [1,2]. As part of DevOps, Infrastructure-as-Code (IaC) [3] is a strategy that "*promotes managing the knowledge and experience inside reusable scripts of infrastructure code, instead of traditionally reserving IT for the manual-intensive labor of system administrators which is typically slow, time-consuming, effort-prone, and often even error-prone*".

IaC adoption has rapidly increased the recent years to speed up the provision and configuration of infrastructural resources using automation, based on practices from software development such as version control and automated testing [3]. On the one hand, these practices allow for a more efficient and repeatable infrastructure code development. On the other hand, frequent changes to the infrastructure permitted by these practices may potentially introduce defects [4] or threat its quality if not adequately supported, raising concerns about IaC quality and the need for instruments to assist developers to quickly evolve and continuously improve infrastructure code, as observed in a recent survey by Guerriero et al. [5] conducted with industrial practitioners.

However, despite its growing interest, only a few works and tools dealt with the quality of IaC. Bent et al. [6] developed a measurement model and a tool to analyze the Puppet code's maintainability. They showed that the measurement model provides quality judgments of Puppet code that closely match the experts' recommendations through structured interviews with experts. Other works focused on the analysis of source code properties to build machine-learning models that evaluate the defectiveness of

* Corresponding author.
E-mail address: s.dallapalma@uvt.nl (S. Dalla Palma).

https://doi.org/10.1016/j.softx.2020.100633

IaC scripts [4] or the occurrence of security smells in infrastructure code, such as the Security Linter for Infrastructure-as-Code scripts [7], initially developed for Puppet. Finally, Ansible Lint[1] helps developers deal with infrastructure code by checking playbooks for practices and behavior that could be improved. Those include formatting and idiom issues, such as trailing whitespace and comparison to an empty string, respectively; tasks issues such as unnamed tasks, and more.

To build a more robust and variegate suite on this line of research, this paper presents AnsibleMetrics, a static source code analyzer for Ansible, the de-facto standard configuration management technology adopted in industry [5]. To foster its adoption, the tool is publicly available as open-source project[2] along with documentation.[3] Although some metrics implemented in AnsibleMetrics might overlap with the checks defined in Ansible Lint, the tools have different goals. The latter focuses on best practices while the former is originally intended to identify source code properties and statistics that can be used as early indicators of faulty infrastructure scripts, potentially leading to expensive infrastructure failure.[4] We deem the two tools could be used in conjunction. Indeed, Ansible Lint identifies violations of best practices defined by rules, while the metrics extracted in AnsibleMetrics could be combined to characterize failure-prone IaC scripts in the scope of Software Defect Prediction [8] or to identify symptoms that indicate wrong style usage or a lousy design, a.k.a. code smells [9].

The remainder of this paper is structured as follows. Section 2 introduces Ansible technology. Section 3 describes the tool's architecture. Section 4 provides illustrative examples to understand the characteristics and use of the proposed tool. Finally, Section 5 concludes the paper and outlines future works.

2. Ansible: an overview

Ansible is a software operations automation engine based on the YAML language. Its purpose is to automate cloud infrastructure provisioning, configuration management, application deployment, and more. Ansible connects nodes as part of a *blueprint* and pushes out scripts called *Ansible modules*, most of which describe the state of the system. Ansible executes such modules that are removed when not needed.

In addition to modules, Ansible *playbooks* orchestrate multiple slices of the infrastructure topology by providing means to precisely control the scalability of the architecture. Playbooks are essential for configuration management and multi-machine deployment. They can declare configurations and launch tasks within one or more *plays* to orchestrate steps of any manual ordered process. A play's goal is to map a group of hosts to some well-defined roles, represented by Ansible *tasks*, which are calls to *Ansible modules*.

Fig. 1 shows an Ansible code snippet representing a playbook that provisions and deploys a website.[5] The Ansible engine configures various aspects such as the ports to be opened on the host container, the user account's name, and the desired database to be deployed. The engine checks that both the Apache server and postgreSQL database are at the latest version and started. This goal is achieved by mapping the hosts (lines 2 and 13) to their respective tasks (lines 8–11, 17–20, 22–25). The modules

```
1  ---
2  - hosts: webservers
3    vars:
4      http_port: 80
5    remote_user: root
6
7    tasks:
8    - name: ensure apache is at the latest version
9      yum:
10       name: httpd
11       state: latest
12
13 - hosts: databases
14   remote_user: root
15
16   tasks:
17   - name: ensure postgresql is at the latest version
18     yum:
19       name: postgresql
20       state: latest
21
22   - name: ensure that postgresql is started
23     service:
24       name: postgresql
25       state: started
26
```

Fig. 1. An Ansible playbook.

`yum` and `service` manage the installed packages and control the active services on the remote hosts, respectively. `name` (i.e., the name of the package and the database) and `state` (i.e., whether present, absent, or otherwise) are parameters of these modules. By composing a playbook of multiple plays, it is possible to orchestrate multi-machine deployments and run specific commands on the machines in the `webservers` group and in those in the `databases` group.

3. AnsibleMetrics: Characteristics and outline

AnsibleMetrics is a Python-based static source code analyzer for Ansible blueprints that helps quantify the characteristics of Infrastructure-as-Code to support DevOps engineers when maintaining and evolving it. It currently supports the 46 source code metrics proposed by Dalla Palma et al. [10] that include the number and the size of plays and tasks, the number of commands, best and bad practices, though others can be derived by combining the implemented ones. The metrics were extracted from the Ansible documentation and the existing scientific literature on object-oriented metrics and IaC measures from Puppet. Please, see the documentation[6] for further details about each metric. The tool is open-source and available on Github and the Python Package Index (PyPI).[7] This section describes the design of AnsibleMetrics, as well as its main APIs.

3.1. Domain object

AnsibleMetrics relies on `AnsibleMetrics` and `LinesMetric`, both providing a single method *count()* to compute the value for a given IaC metric. In particular

- `AnsibleMetric` is an abstract class extended by all the classes responsible for computing metrics for a given Ansible construct (e.g., count the number of plays, tasks, modules, etc.). It takes a plain YAML file as input and stores

[1] https://github.com/ansible/ansible-lint (Accessed September 2020).
[2] https://github.com/radon-h2020/radon-ansible-metrics.
[3] https://radon-h2020.github.io/radon-ansible-metrics/.
[4] https://cloudcomputing-news.net/news/2017/oct/30/glitch-economy-counting-cost-software-failures/ (Accessed September 2020).
[5] Adapted from Ansible documentation: https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html (Accessed September 2020).
[6] https://radon-h2020.github.io/radon-ansible-metrics/.
[7] https://pypi.org/project/ansiblemetrics/.

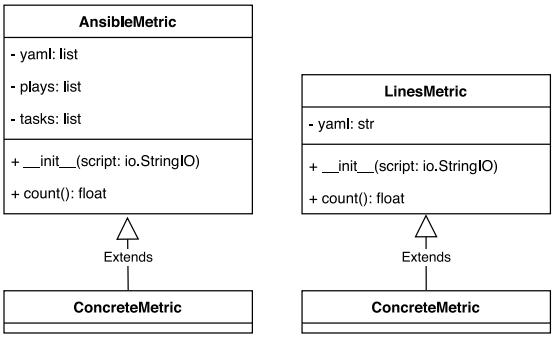


Fig. 2. Diagram of the two base classes of AnsibleMetrics.

it as a list of dictionaries (note that the dictionary is the datatype provided by the library `PyYAML` used to parse it). In addition, it extracts the *plays* and *tasks* from the input script and stores them as list of dictionaries in the class attributes `plays` and `tasks`, respectively. This step facilitates the implementation of the metrics: no further check concerning the structure of `plays` and `tasks` is needed.

- `LinesMetric` is an abstract class responsible for computing language-agnostic metrics related explicitly to lines of code (e.g., number of executable, commented, and blank lines). Similar to `AnsibleMetrics`, it takes a plain YAML file as input. Such a file is stored as a string to allow for parsing comments and blank lines ignored by `PyYAML`.

Both classes validate the YAML file in their *__init__()* method and raise a *ValueError* whenever the file is empty or is not well-formatted. Therefore, the extended classes (`ConcreteMetric` in Fig. 2) can safely assume the file is a valid YAML when parsing it and implementing the logic of the method *count()*.

3.2. Software architecture

AnsibleMetrics presents a two-level software architecture as schematized in Fig. 3(a). The higher level is composed of the user interfaces (GUI and Command-line) and code APIs. They provide the main entry points for third-party applications that need to quantify their Infrastructure-as-Code characteristics.

The GUI[8] consists of a Visual Studio Code (VSC) extension available on the VSC Marketplace.[9] The CLI takes the path of an Ansible blueprint or a directory of blueprints; optionally, it saves the report in *json* format to the path specified by the user. The *json* report consists of an object containing *(key, value)* pairs, where the *key* is the metric name and the *value* is the computed value, similar to the following:

```
{
  "avg_task_size": <integer>,
  "num_tasks": <integer>,
  "num_plays": <integer>,
  "text_entropy": <float>
}
```

The APIs allow for two main usage scenario:

1. `MetricExtractor` (Fig. 3(b)) programmatically runs all the implemented metrics. It returns a *json* object parsable by the developer, as for the command-line interface. The CL relies on `MetricExtractor`.
2. In turn, `MetricExtractor` relies on the classes that specialize `AnsibleMetric` and `LinesMetric`. A DevOps engineer can programmatically run a given metric singularly depending on the problem at hand, without having to extract all the metrics at once. In that case, the value returned by the metric is the one returned by its method *count()*, that is, an integer or floating number.

4. AnsibleMetrics: Usage examples

For illustrative purpose, let us consider the example in Fig. 1. To analyze this blueprint, the user has to install the `ansiblemetrics` library from PyPI with the command:

```
pip install ansiblemetrics
```

or, alternatively, from the source code root folder:

```
pip install .
```

The user can run the command within a terminal or import the `ansiblemetrics` module in a code snippet.

4.1. Command-line usage

Assuming that the example in Fig. 1 is named *playbook1.yml* and is located within the folder *playbooks* as follows:

```
playbooks/
 |- playbook1.yml
 |- playbook3.yml
 |- playbook3.yml
```

and assuming the user's working directory is the *playbooks* folder, then it is possible to extract source code characteristics from that blueprint by running the following command:

```
ansible-metrics playbook1.yml --dest report.json
```

For this example, the *report.json* will result in

```
{
  "filepath": "playbook1.yml",
  "avg_play_size": 10,
  "avg_task_size": 4,
  "lines_blank": 4,
  "lines_code": 20,
  "num_keys": 20,
  "num_parameters": 6,
  "num_plays": 2,
  "num_tasks": 3,
  "num_tokens": 50,
  "num_unique_names": 3,
  "num_vars": 1,
  "text_entropy": 4.37
}
```

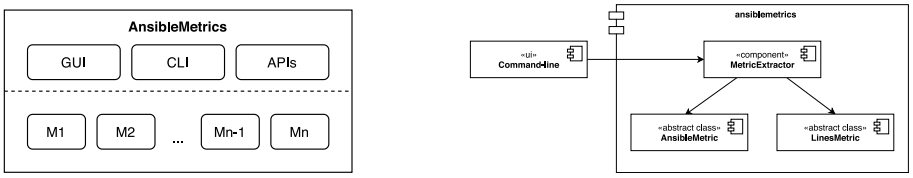
Please note that for the sake of space, we reported only the non-zero metrics in this example, while the report contains all metrics by default. To omit the zero metrics in the report, the user has to pass the parameter `--omit-zero-metrics`.

The user can pass as input the path to the folder containing them to avoid running the command multiple times for multiple blueprints. For example:

```
ansible-metrics . --dest report.json
```

[8] Source available on Github at https://github.com/radon-h2020/radon-ansible-metrics-plugin.

[9] https://marketplace.visualstudio.com/items?itemName=radon-h2020.ansiblemetrics.



(a) A logical schema of the tool.

(b) A component diagram of the AnsibleMetrics architecture.

Fig. 3. Architecture overview.

```python
from io import StringIO
from ansiblemetrics.metrics_extractor import extract_all
from ansiblemetrics.general.lines_code import LinesCode
from ansiblemetrics.playbook.num_plays import NumPlays

script = StringIO("") # To fill with the example in Figure 1

all_metrics = extract_all(script)
loc_count = LinesCode(script).count()
plays_count = NumPlays(script).count()

print('All metrics:', all_metrics)
print('Lines of code:', loc_count)
print('Number of plays:', plays_count)
```

Fig. 4. Example of Python usage of AnsibleMetrics.

In this case, the *report.json* will consist of an array of *json* objects of the type:

```
[
  {
    "filepath": "playbook1.yml",
    "metric1": <value>,
    ...
    "metricN": <value>
  },
  ...
  {
    "filepath": "playbook3.yml",
    "metric1": <value>,
    ...
    "metricN": <value>
  }
]
```

In the previous examples, the *report.json* files are saved in the user's working directory, so that the *playbooks* folder looks as follows:

```
playbooks/
 |- playbook1.yml
 |- playbook3.yml
 |- playbook3.yml
 |- report.json
```

However, the user can specify a different path where to save the report for later usage. Please consider that where multiple files with various extensions are present, running the command on the entire folder will only analyze Ansible blueprints.

4.2. Python module usage

The previous section showed how the user could exploit the command-line interface to extract metrics from an Ansible blueprint. However, there could be the need to obtain a metric or a group of them programmatically. In this scenario, the user can efficiently address this task by importing the `ansiblemetrics` module in her code snippet. More specifically, it is possible to import the `ansiblemetrics.metrics_extractor` module to extract all the metrics at once and the `ansiblemetrics.playbook.<metric>` or `ansiblemetrics.general.<metric>` (where `<metric>` has to be replaced with the name of the desired metric) module to compute a specific metric. The difference between the latter two modules lies in the fact that the *playbook* module contains metrics specific to playbooks (e.g., number of plays and tasks). In contrast, the *general* module provides metrics that can be generalized to other languages (e.g., lines of code).

Fig. 4 exemplifies these scenarios. In that code snippet, `all_metrics` will store a *json* object as for the example in Section 4.1, while the `loc_count` and `plays_count` will contain the values 20 and 2, respectively.

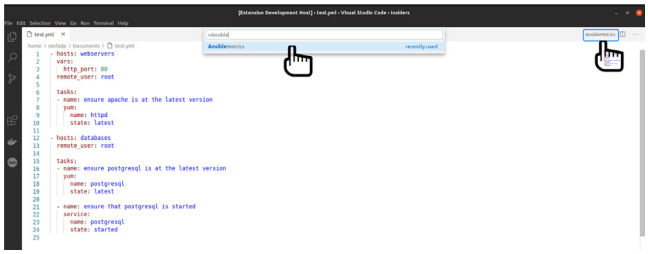


Fig. 5. How to run the plugin.

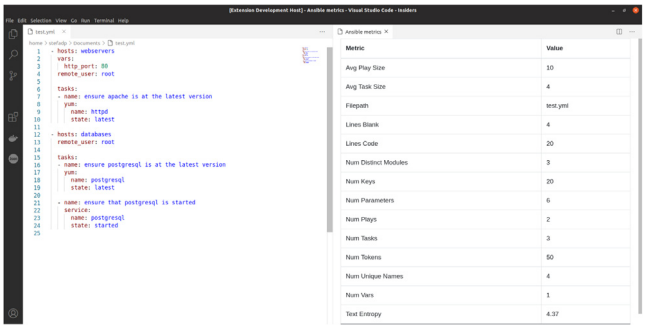


Fig. 6. Example of report.

4.3. GUI usage

Figs. 5 and 6 depict two parts of the Graphical User Interface. The former shows how to run the plugin while the second an example of the report. The GUI ease using the tool when using it through editors that support Visual Studio Code (VS Code) extension. First, the user has to install the `ansiblemetrics` dependency, as described in the previous section. It is then possible to install the extension by directly looking for it on the VS

```yaml
- name: Ensure ansiblemetrics is at its latest version
  pip:
    name: ansiblemetrics
    state: latest

# Ensure that foo.yaml exist
- name: Retrieving file status
  stat:
    path: /playbooks/foo.yaml
  register: foo

- name: Run ansible-metrics if foo.yaml exists
  shell:
    chdir: playbooks/
    cmd: 'ansible-metrics foo.yml --dest report.json'
  when: foo.stat.exists is defined and foo.stat.exists
```

```
# Ensure ansiblemetrics is at the latest version
packages {'ansiblemetrics':
  ensure   => 'latest',
  provider => 'pip',
}

# Ensure that foo.yaml exist
file {'foo.yaml':
  ensure => 'file',
  path   => '/playbooks/foo.yaml',
}

exec {'run ansible-metrics':
  cwd     => 'playbooks/',
  command => 'ansible-metrics foo.yml --dest report.json',
}
```

Fig. 7. Ansible tasks (left) and Puppet resources (right) to install AnsibleMetrics from PyPIP and run it.

Marketplace[10] or launching the VS Code Quick Open (`Ctrl+P`), paste the following command, and press enter:

```
ext install radon-h2020.ansiblemetrics
```

Finally, it is possible to run the extension in two ways: (1) by launching the VS Code commands palette (`Ctrl+Shift+p`) and typing *AnsibleMetrics* or (2) by clicking on the `AnsibleMetrics` button on the top-right corner of the editor. Please note that the latter will be visible only for YAML-based Ansible files.

5. Conclusions

In this paper, we presented AnsibleMetrics, a novel tool to measure the software quality of Ansible blueprints. AnsibleMetrics aims at making a step forward to support DevOps engineers when developing and maintaining infrastructure code.

We illustrated its architecture and examples of usage through a command-line and graphical user interface. Although a GUI is currently available only as a VS Code extension, it can be easily integrated into other environments. More specifically, a version is planned for the Eclipse Che.[11] Indeed, starting with Eclipse Che 7.15, VS Code extensions can be effortlessly installed to extend the functionality of a Che workspace.[12]

Although we provide the user with different modalities of interaction with the tool, additional ones (e.g., exporting as HTML report or a web application) are planned for future work. We will also improve the tool by providing actionable suggestions to the programmer by highlighting the code snippet that violates a specific metric of choice. The final goal is to highlight the parts of code that require more auditing using the extracted metrics to capture Infrastructure-as-Code aspects that could act as a proxy for defect prediction.

We are confident that the proposed tool will help developing measurement models for the quality of Ansible blueprints, in particular, and IaC in general. We also deem the tool appropriate and usable in practice, although a validation with experts is part of our near research agenda.

Although we focus on Ansible, the most used language for IaC, most of the metrics are easily extendible to other languages, such as Puppet.[13] One example is listed in Fig. 7. The two code snippets are analogous: they ensure that the latest version of `ansiblemetrics` is installed, and run the tool via command-line on a file called *foo.yaml* present in the folder *playbooks*. In this example, text-based metrics, such as *line of code and comments*, *text entropy* and *number of tokens*, can be easily extracted from both scripts; similarly, other metrics can be easily adapted. The Ansible code in Fig. 7 (left) has *three* tasks (lines 1, 7, and 12), each of them calling a module[14] (*pip*, *stat* and *shell*, respectively). Here, the metric *number of modules = 3*. *Modules* are fundamental in Ansible as they provide units of code directly executable on remote hosts or through playbooks. Similarly, the Puppet code in Fig. 7 (right) consists of *three* resources[15] (lines 2, 8, and 13). Puppet's *resources* are the fundamental unit for modeling system configurations. A Puppet resource can be considered analogous to an Ansible module. For example, the metric *number of modules = 3* can be translated into *numbers of resources = 3*. More specifically, it can be generalized to other languages in something similar to *number of fundamental units*.

Other metrics apply as well. Both scripts in Fig. 7 have *number of file exists = 1* and *number of commands = 1*. The former metric counts the number of times a module or resource check the existence of a file, directory, or symbolic link. In Ansible, this is possible through the module *stat* while in Puppet, it is possible through the *ensure* property. The latter metric counts the number of external commands. Different modules allow the execution of these commands in Ansible. Among them, the *shell* module. Puppet, instead, allows executing external commands via the resource type *exec*. In addition, Fig. 7 (left) has *number of parameters = 5* (lines 3, 4, 9, 14, and 15), while Fig. 7 (right) has *number of parameters = 6* (lines 3, 4, 9, 10, 14 and 15). Module's parameters (or arguments) in Ansible describe the desired state of the system. Similarly, Puppet's attributes describe the desired state of a resource; each attribute handles some aspect of the resource.

To conclude, AnsibleMetrics can compute several code metrics that can be generalized to other languages. However, at the current stage, it does not provide any further analysis. As future work, we are implementing a new visualization to help developers identify problematic metrics, which can be perceived as potential proxies to identify symptoms that indicate wrong style usage or a lousy design.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work is supported by the European Commission grant no. 825040 (H2020 RADON).

[10] Available at: https://marketplace.visualstudio.com/items?itemName=radon-h2020.ansiblemetrics.

[11] https://www.eclipse.org/che/.

[12] https://www.eclipse.org/che/docs/che-7/using-a-visual-studio-code-extension-in-che/ (Accessed September 2020).

[13] https://puppet.com/.

[14] https://docs.ansible.com/ansible/latest/user_guide/modules_intro.html (Accessed September 2020).

[15] https://puppet.com/docs/puppet/6.18/lang_resources.html (Accessed September 2020).

References

[1] Bass L, Weber I, Zhu L. DevOps: A software architect's perspective. Addison-Wesley Professional; 2015.

[2] Artac M, Borovssak T, Di Nitto E, Guerriero M, Tamburri DA. DevOps: introducing infrastructure-as-code. In: 2017 IEEE/ACM 39th international conference on software engineering companion (ICSE-C). IEEE; 2017, p. 497–8.

[3] Morris K. Infrastructure as code: managing servers in the cloud. " O'Reilly Media, Inc."; 2016.

[4] Rahman A, Williams L. Characterizing defective configuration scripts used for continuous deployment. In: 2018 IEEE 11th international conference on software testing, verification and validation (ICST). IEEE; 2018, p. 34–45.

[5] Guerriero M, Garriga M, Tamburri DA, Palomba F. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In: 2019 IEEE international conference on software maintenance and evolution (ICSME). IEEE; 2019, p. 580–9.

[6] Van der Bent E, Hage J, Visser J, Gousios G. How good is your puppet? an empirically defined and validated quality model for puppet. In: 2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER). IEEE; 2018, p. 164–74.

[7] Rahman A, Parnin C, Williams L. The seven sins: Security smells in infrastructure as code scripts. In: Proceedings of the 41st international conference on software engineering; 2019. p. 164–75.

[8] Hall T, Beecham S, Bowes D, Gray D, Counsell S. A systematic literature review on fault prediction performance in software engineering. IEEE Trans Softw Eng 2011;38(6):1276–304.

[9] Fowler M. Refactoring: improving the design of existing code. Addison-Wesley Professional; 2018.

[10] Dalla Palma S, Di Nucci D, Palomba F, Tamburri DA. Toward a catalog of software metrics for infrastructure code. J Syst Softw 2020;170(0164–1212).