Received: 29 September 2021 | Revised: 1 April 2022 | Accepted: 16 May 2022
DOI: 10.1002/smr.2480

SPECIAL ISSUE - TECHNOLOGY PAPER

Journal of Software: Evolution and Process WILEY

Predictive maintenance of infrastructure code using "fluid" datasets: An exploratory study on Ansible defect proneness

Giovanni Quattrocchi[1] | Damian Andrew Tamburri[2]

[1]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy

[2]Jeronimus Academy of Data Sciences, Eindhoven University of Technology, 's-Hertogenbosch, The Netherlands

Correspondence
Giovanni Quattrocchi, Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy.
Email: giovanni.quattrocchi@polimi.it

Abstract

This work consolidates and compounds previous investigations in recognizing defects for infrastructure-as-code (IaC) scripts using general software development quality metrics with a focus on defect severity but adding to previous work an explorative look at creating datasets, which may boost the predictive power of provided models—we call this notion a *fluid dataset*. More specifically, we experiment with 50 different metrics harnessing a multiple dataset creation process whereby different versions of the same datasets are rigged with auto-training facilities for model retraining and redeployment in a DataOps fashion. At this point, with a focus on the Ansible infrastructure code language—as a de facto standard for industrial-strength infrastructure code—we build defect prediction models and manage to improve on the state of the art by finding an F1 score of 0.52 and a recall of 0.57 using a Naive–Bayes classifier. On the one hand, by improving state-of-the-art defect prediction models using metrics generalizable for different IaC languages, we provide interesting leads for the future of infrastructure-as-code. On the other hand, we have barely scratched the surface on the novel approach of fluid-datasets creation and automated retraining of Machine Learning (ML) defect prediction models, warranting for more research on the same direction in the future.

KEYWORDS
defect prediction, DevOps, fluid datasets, infrastructure code

1 | INTRODUCTION

With the "need for speed" being more and more important in the current IT market, the software development cycle is becoming shorter everyday.[1] Because of this more agile approach on software development and IT operation teams, both teams are more and more cooperating as a DevOps team. A lot of good software engineering practices are adopted by these teams in the form of Infrastructure as Code (IaC): code-versioning, code-revision management, and so forth,[2] but—according to Gartner— infrastructure code failure and downtimes can cost $5600 per minute.[3] According to Kief Morris,[4] IaC can be defined as follows:

> "a declarative or procedural code to provision and/or configure software-defined infrastructure. It is an approach to infrastructure automation based on software development practices, emphasizing consistent, repeatable routines for changing systems and their configuration."

In essence, IaC helps practitioners in automating the management of the infrastructure by mapping each maintenance task with a dedicated script that is unambiguous, versionable, and re-executable. However, being *code*, IaC is subject to bugs and defects that could significantly hinder the operations of a software system.

According to Rahman et al.,[5] IaC is a relatively new research area with most research done on IaC frameworks, their usage as well as their defect proneness. Similarly, other recent publications have been related to empirical meta-studies. Rahman et al[5] fail to find studies on IaC defects and security flaws and observes a need for this type of study while also highlighting the shortage of data concerning specific types of defects and issues thereof. Similarly, previous studies of the same authors have shown that defects can occur in IaC scripts, resulting in costly system outages.[6]

In summary, although recent research is starting to take IaC into account for software quality analyses, there is a shortage of both data and increased risks connected to IaC outages. Stemming from this fact and drawing from typical bagging[7] and bootstrapping[8] practices in machine learning and computational intelligence,[9] we conjecture that different versions of the same existing datasets in IaC can be created to boost the predictive performances of available IaC defect-predictive models, with a focus on predicting the defect *suspiciousness* of areas of code, rather than specific code instances—we refer to this notion as a *fluid* dataset; that is, a dataset that is created and recreated according to prespecified code data properties and characteristics drawn from the target modelling problem's features. To address the aforementioned conjecture, we enact an exploratory ML-based investigation focusing on predictive modelling of defects in the Ansible language. Our goal is to answer the following research questions:

- **RQ1**: How well can we classify IaC scripts as defective using software development quality metrics within *fluid* datasets?
- **RQ2:** Can changes in code be of predictive value for the defectiveness of the infrastructure code?
- **RQ3:** How well can we generalize the fluid-data quality prediction of an IaC script to other IaC languages?

To answer **RQ1**, we train several machine learning models to classify Ansible Playbooks as *defect* or *sound* using metrics from the state of the art.[10] We scrape multiple GitHub repositories and extract their IaC scripts as a data set for model training and validation. We acquire labeled data by searching through the commit history of the repositories, and we assume that a commit containing the keyword `FIXED` fixes a bug.[*] We retrieve the code before and after the commit and label them as *defect* and *sound*, respectively.

To answer **RQ2**, we use the labeled data from **RQ2** by trying to correctly predict the code before the commit as defect and after the commit as sound.

To answer **RQ3**, we look at the generalizability of the identified important metrics. In this research, we focus mainly on Ansible scripts. However, by taking an extra look at generalizable metrics, we can extend our scope to other IaC languages like Chef, Puppet, and Tosca as well.

In the following sections, we first discuss the available literature on software development quality metrics, defect prediction, and DevOps in Section 2. In Section 3, we discuss our experimental configuration, our approach on generating data and metrics, and the clustering and prediction techniques used to come to our results. In Section 4, we give a brief discussion of all the generated metrics and analyze their behavior in the data set. We discuss the clustering results in Section 5 and the prediction results in Section 6. Our research questions are answered in Section 7. We conclude our research in Section 8.

2 | BACKGROUND AND RELATED WORK

This section first describes Infrastructure as Code with an example in Ansible that helps the reader to grasp the main concept of the domain. Second, to reach our goal of using traditional software quality metrics to find defects in IaC scripts, we review the history and the current state of these metrics to find metrics that can be used, and to gain knowledge of the effect of these metrics on the quality of software. Third, we give an overview of existing defect prediction methods. Including data generation, metrics, and defect prediction models. Fourth, we give an overview of the workflow and problems of development and operations teams in a company. Fifth, we analyze existing research on defect prediction in DevOps. We conclude by summarizing the key findings from the literature and how we use them in this manuscript to answer our research questions.

2.1 | Infrastructure as code

While cloud computing materialized the idea of a programmable backend with virtually infinite resources, it also increased the need of a more rigorous, deterministic, and less error-prone approach to the provisioning, configuration, and management of (virtual) resources. This need is the main enabler of the DevOps paradigm[2] that binds any management task with a corresponding IaC script that is versionable, executable, and unambiguous. Several languages and platforms have been realized in the last few years, each focusing on specific characteristics of infrastructure management ranging from virtual machine (e.g., Cloudify and Terraform) and container (e.g., Docker Swarm and Kubernetes) orchestrators to configuration management tools (e.g., Ansible, Chef, and Puppet).

Among other tools, Ansible[†] is recently gaining traction[‡] for its simple and agent-less architecture, and we use it as an exemplary technology in the rest of this subsection. Ansible is an automation engine that allows users to provision cloud resources and configure the infrastructure and applications using the YAML language. Ansible allows to write two main types of artifacts: *modules* and *playbooks*.

Ansible connects to infrastructure nodes (e.g., provisioned virtual machines) and execute modules to change their configuration (i.e., each module executes a single management operation). The execution of a module is called a task. The orchestration of tasks is made possible by playbooks. Playbooks allow to deploy and configure multiple nodes and orchestrate and synchronize the execution of tasks on a slice of or the whole infrastructure topology.

As an example,[11] Figure 1 shows an Ansible playbook that provisions and deploys a website composed of a set of web servers and a PostgreSQL database. Lines 2–5 identify the host type *webservers*, while line 7 declares the corresponding set of tasks. Lines 8–11 defines a task that the latest version of the Apache web server is installed. The task executes module *yum* with a given name and desired state (i.e., *latest*). Lines 13–14 identify the *databases* host type. Lines 17–20 and 22–25 declare two dedicated tasks, respectively. The former ensures the installation of the latest version of PostgreSQL (again with module *yum*), while the latter guarantees that PostgreSQL is started on each database node (with module *service*). With more complex scripts, it is possible to synchronize the execution of the different tasks, order their execution and create conditional executions.

2.2 | Software development quality metrics

Li et al[12] study 31 different software metrics and find that metrics based on program size are useful for evaluating software quality. Tang et al[13] identify three types of faults correlated with object-oriented design metrics. The first two types identified are object oriented, the third type is not. The metrics mentioned can all be related to either class complexity, inheritance, and coupling.

Xenos et al[14] provide an overview of existing software development quality metrics that can be used for object oriented programming (OOP). The authors present a set of traditional metrics that can be applied to OOP and a set of metrics specifically designed for OOP. According to the authors, the latter cannot be applied to other programming styles than OOP.

Yu et al[15] define a software metric as a *measurement to quantify characteristics or attributes of a software entity*. The authors divide software metrics into three categories: Quality of source codes, development processes, and accomplished applications. Software complexity is said to be focusing on the quality of source code, divided into three categories: Essential complexity, selecting complexity, and incidental complexity. The

```
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
```

FIGURE 1 An example of an Ansible script

essential and selecting complexity are set by the problem being solved and the programming language, whereas the incidental complexity is set by the implementation.

According to Keshavarz et al,[16] complexity is an important factor to affect code quality. The authors analyze different factors that are important to software complexity. The number of inputs, outputs, files, external interfaces, and user inquiries are mentioned. The authors also mention the Halstead Complexity Metric (HCM), which is based on the number of unique operators and operands, and the number of all operators and operands. The McCabe Cyclomatic Complexity Metric is computed by counting the number of decisions in a program, which is related to the number of IF/ELSE statements. The authors however do warn for the difficulty of computing the HCM and the McCabe complexity.

Rashid et al[17] study the impact of software metrics on software quality, the authors conclude that the reliability of a program decreases once the complexity increases: *Object oriented metrics prove to be very good in predicting complexity fast and, therefore, these metrics enhance software quality*.

The metrics studied by Li et al[12] are mostly measurements of different properties in a script, most of which are applicable to each type of language. Tang et al[13] concludes that a large method, containing more code, introduces more defects than a small method, motivating to look at the size of a script and its contents. Because of the findings of Rashid et al,[17] we try to create a similar metric applicable to IaC for each metric used by Xenos et al.[14] Like the work of Yu et al,[15] we focus on the quality of source code. Because of Keshavarz et al.'s[16] warning for the difficulty of computing the HCM and McCabe complexity, we start by focusing on the easier to compute metrics that they are related to, like the number of operators and operands, and the number of IF/ELSE statements.

Being full-fledged code, traditional software metrics can be applied to IaC as is. Della Palma et al[11] proposed an initial refinement of traditional software metrics into ones dedicated to IaC that takes into account, for example, not only the numbers of code block in a script but also the number of Ansible tasks or distinct modules. However, the study of a broader taxonomy of IaC quality metrics is considered out-of-scope for this paper and it is considered future work.

2.3 | Defect prediction

Defect prediction[18] is an important research field of software engineering. Several approaches have been presented in the literature including ones that estimate the number of defects in a software program,[19] others[20] that find “associations” among defects (i.e., they detect additional problems starting from a defected piece of code), and ones[21] that classify the defect-proneness of software components (as the ones discussed in this manuscript).

Defect prediction models can be used to control (semi-)automatically the software integration cycle: If number of defects or the defect-proneness of a software artifact is too high, its deployment in production can be delayed or subject to further inspection. Defect prediction models can also be used as tools to guide software reviews (e.g., pull requests) helping the reviewer in discovering bugs and their root causes (e.g., through defects associations) or to highlight the need of a refactoring action. Finally, they can be utilized for prioritizing unit and integration testing (roughly, the more defects detected the more tests are needed).

Several defect prediction models, as the one presented herein, can be complemented with an explainability layer that transcends the detection task and helps practitioners understanding *why* a software component is defected.[22] This kind of extensions deserves a dedicated focus, and their integration in the presented model is considered out-of-the scope of this work.

When creating a defect prediction model, the first step is to generate data from software archives. Version control systems, e-mail archives, and issue tracking systems are examples of these archives. The second step is to extract metrics or features from the generated data, the third step is to label the data as either sound, buggy, or defect. An option is to pre-process the data, by using feature selection, data normalization, or noise reduction. At last, a prediction model can be trained that classifies whether a program contains bugs or not.[23]

Wan et al[24] give a recent overview of defect prediction models. The authors identify the following categories:

1. Prediction models and learning approaches
2. Metrics in defect prediction models
3. Cross-project defect prediction
4. Imbalance, mislabeling, bias and noise
5. Granularity of defect prediction
6. Privacy
7. Performance evaluation

We follow a similar strategy to Nam.[23] Our study is placed into categories 1 and 2 of Wan et al.[24]

2.3.1 | Data generation

Vendome et al[25] mine the commit history of 16,221 open source Java Applications from GitHub by first mining a list of projects hosted on GitHub using the GitHub API. Similarly, Reyes et al[26] provide a study collecting available research on GitHub. The study provides a set up for retrieving data from GitHub and gives insight on the use of keywords. Horton et al[27] present *Gistable*, a database and framework using Github's gist system, containing 10,259 code snippets. Schermann et al[28] provide 100,000 unique Dockerfiles in over 15,000 Github repositories to answer questions about typical programming languages and defects.

The mentioned researches focus on different coding styles and languages, but their research can prove helpful in crawling GitHub repositories and generating data.

2.4 | Defect prediction in DevOps

Schwarz et al[29] use code smells to assess the quality of Puppet code. They also find that smells from other technologies are transferable. New code smells that are introduced and applied to Chef are as follows: Long resources, using too many attributes, not using comments, the Law of Demeter (*each unit should only have limited knowledge about other units*), using hyphens, and having empty default values. We note that the last two smells are derived directly from the Chef style guide.

Rahman et al[30] tries to identify characteristics of Puppet scripts that correlate with defects and violate security and privacy objectives. 14 IaC characteristics that correlate with defects are found.

Rahman et al[6] also do a first step toward predicting defects in IaC scrips, by using textual features to train a random forest (RF) classifier that classifies defective Puppet scripts.

Borg et al[31] presents an implementation of the SZZ algorithm to identify bug-introducing software changes. The authors find an F1 score of $[0.10-0.15]$, a recall of $[0.13-0.20]$, and a precision of $[0.07-0.12]$ when predicting bugs in Jenkins code.[§]

2.5 | Technical landscape

Infrastructure as code is becoming a key component in the management of applications because it facilitates, speeds up, and automates recurring nonfunctional operations. IaC materializes any of these tasks with a dedicated script that is versionable and re-executable. Being code, IaC scripts (e.g., Ansible, Chef, and Puppet) can be analyzed through software quality metrics and prediction models can be created in order to help practitioners detect defects. While some approaches have been presented on this matter, the literature on IaC defect proneness is still in its infancy calling for more research effort.

3 | METHODOLOGY

3.1 | Research problem

The technical space of infrastructure code—intended as the set of technological assets, languages, and tools potentially employable for infrastructure code design, development, and management—is quite extensive and therefore a need arises to produce metrics as well as varied perspectives on their usage to *quantify* and *predictively model* infrastructure code.

While some quantification mechanisms are starting to emerge (as described in Section 2), they tend to be technology-specific and do not account for the individual varieties of each technology, for example, varying the features and characteristics to be quantified depending on the target (set of) technologies.

3.2 | General overview and experimental setup

The general approach for this work—tailored from previous studies that paved the way to the topic[23]—is recapped in Figure 2.

In essence, to answer **RQ1**, we train classification models on two different target variables. The first target variable is the clustering result of **RQ1**, the second target variable is the outlier result. The classification models are discussed in Section 3.7.

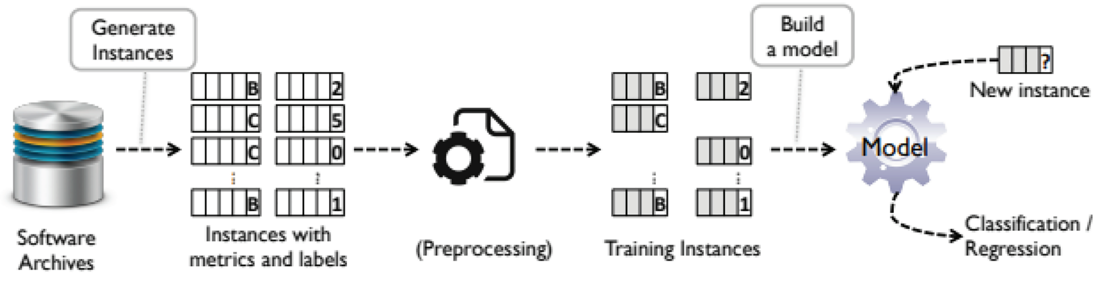

FIGURE 2 Common process of software defect prediction, tailored from previous work;[23] the same and identical approach was followed for this work

TABLE 1 Results of GitHub scraping

	Ansible	Chef	Puppet
Total repositories	428	216	224
Number of files	25,034	1,071	2,682
Number of authors	246	85	402
Avg. stars	301	174	94

To answer **RQ2**, we use the classification models trained to answer **RQ1**. We count the number of times an instance from our test set is predicted *defect* before the commit, and *sound* after the commit. We use the same classification models that are used to answer **RQ1**; they are discussed in Section 3.7.

To answer **RQ3**, we discuss the metrics and their generalizability in Section 4.1.

Essentially, to complete this research, we first scrape GitHub repositories to find Ansible, Chef, and Puppet files. After finding that Ansible data returns the most data files, we converge our research to focus solely on Ansible files.

Second, we consider a total of 50 different metrics using the Ansible documentation and existing literature on software development quality metrics. For each metric, we estimate its generalizability for other IaC languages. By combining existing research software development quality metrics and existing research on the other popular IaC languages with the Ansible documentation, we make an educated guess about metrics that are relevant for the complexity of Ansible scripts and therefore influence the probability of defects present in the script.

Third, we create a dataset of measurements of each metric for each Ansible script (see Section 6.4 for more information about the tool that we created to accomplish this task). Some of the metrics are correlated by design. Think of the number of blank lines of code in a script and the *relative* number of blank lines of code: The number of blank lines of code compared to the total lines of code. To compare for the different effects of these metrics in bug prediction, we create eight separate datasets: The first containing all the metrics (AM), the second containing only the absolute metrics ("number of blank lines of code") (ABS), the third containing only the relative counterparts of these absolute metrics (REL), and the fourth containing no metrics with a correlation higher than 0.7. Each dataset is considered with and without outliers.

These datasets are used for outlier detection and clustering. We use nine different outlier detection algorithms that can be found in Table 3. We use an independent ensemble framework with a maximization method for score combination. Outliers can be interesting for our research because an outlier, by definition, does not behave according to the rest of the population. We can use them to identify new smells in Ansible scripts and use these smells to create a defect prediction algorithm. If the set of outliers indeed consists of a lot of defective scripts, we can also use the set for training our prediction model.

Clusters are created for the same reasons as outliers are detected: By analyzing the resulting clusters, we find Suspicious Clusters, as defined in Section 1. We use an ensemble clustering framework using random projection to generate ensemble members, spectral graph partitioning as a consensus function, and silhouette, Davies Bouldin, and Calinski Harabasz scores as evaluation metrics.

Fourth, we train classification models on the resulting clusters. We gain insights on metrics that are relevant for the suspicious clusters. The methods used can be found in Section 3.6.

Fifth, we train random forest, decision tree, and Naive–Bayes classifiers, as suggested by Rahman[30] and Hammouri et al,[32] on the clustering and the outlier detection results. We train each classifier on each dataset and compare its results. We also train each classifier on each dataset, using only the most important metrics, classified in the previous step.

3.3 | Data gathering

We use a GitHub scraper to extract multiple repositories related to IaC from Github.[33] To ensure some quality, each repository should have at least 5 stars and has to be updated recently (within six months since the day of scraping). Using the keywords *Ansible, Chef, IaC, Infrastructure as*

TABLE 2 Infrastructure code metrics considered in this study; name and abbreviation of each metric are reported in columns 1 and 2, while their respective factor of generalizability is reported in column 3

Abbreviation	Name	Type
ATSS	Average task size	Semi-specific
BLOC(*_relative*)	Blank lines of code	General
CLOC(*_relative*)	Commented lines of code	General
DPT	Depth	General
ETP	Entropy	General
LOC	Lines of code	General
NBEH(*_relative*)	Number of blocks error handling	Semi-specific
NBL	Number of blocks	Specific
NCD	Number of conditionals	General
NCMD	Number of 'command' commands	Semi-specific
NCO	Number of comparison operands	General
NDK(*_occurrences*)	Number of deprecated keywords	Semi-specific
NDM(*_occurrences*)	Number of deprecated modules	Semi-specific
NEMD(*_relative*)	Number of external modules	Semi-specific
NFL	Number of filters	Semi-specific
NFMD(*_relative*)	Number of fact modules	Semi-specific
NICD	Number of inline conditionals	Specific
NIERR	Number of ignore_errors	Specific
NIMPP	Number of import_playbook	Semi-Specific
NIMPR	Number of import_role	Semi-Specific
NIMPT	Number of import_tasks	Semi-Specific
NINC	Number of include	Semi-Specific
NINCR	Number of include_role	Semi-Specific
NINCT	Number of include_tasks	Semi-Specific
NINCV	Number of include_vars	Semi-Specific
NKEYS	Number of keys	General
NLK	Number of lookups	Semi-specific
NLO	Number of logic operands	General
NLP	Number of loops	Semi-specific
NMD	Number of modules	Semi-specific
NMO	Number of math operands	General
NNNV(*_relative*)	Number of names without variables	Semi-Specific
NNWV(*_relative*)	Number of names with variables	Semi-Specific
NSCM	Number of suspicious comments	General
NSH	Number of shell commands	Semi-Specific
NTKN	Number of tokens	General
NTNN(*_relative*)	Number of tasks without a name	Specific
NTS	Number of tasks	Semi-Specific
NTUN(*_relative*)	Number of tasks with a unique name	Semi-Specific
NTVR	Number of task variables	Semi-Specific
NUN(*_relative*)	Number of unique names	General

Code, *Puppet*, and *TOSCA*, we find a total of 880 repositories, written by as many as 402 different authors per technology (for a total of 733 authors involved as indirect subjects of our study) and containing 28,787 files. The average number of stars for each repository is 215.5, with an average of 231 pull requests. The most popular programming languages for each keyword are as follows: Python for Ansible, and Ruby for Chef and Puppet. The results can be found in Table 1. We find no more than 30 repositories for the keywords *IaC*, *Infrastructure as Code*, and *TOSCA* combined. Therefore, we consider them to be irrelevant for this work and decide to only include *Chef*, *Ansible*, and *Puppet* results. The high number of Ansible scripts together with the little research that has already been done on Ansible as compared to Chef and Puppet motivated us to focus the rest of our research on the Ansible technology.

3.4 | Metrics and fluid datasets overview

According to the documentation, Ansible Playbooks *describe a policy you want your remote systems to enforce, or a set of steps in a general IT process. Playbooks can be used to manage configurations of and deployments to remote machines.*[34] Based on this documentation and on the existing literature on traditional software development quality metrics, we derive 50 metrics drawn from previous work[10] to extract from Ansible playbooks that might provide an indication for defective IaC scripts. All metrics are divided in seven categories, essentially a *fluid* dataset, with six degrees of freedom:

1. **General complexity:** *LOC*, *BLOC*, *CLOC NTKN*, *DPT*, *NKEYS*, *ETP*,
2. **Tasks and modules:** *ATSS*, *NTS*, *NTVR*, *NMD*, *NFMD*,
3. **External:** *NINC*, *NINCT*, *NINCV*, *NINCR*, *NIMPT*, *NIMPR*, *NEMD*, *NLK*, *NSH*,
4. **Naming conventions:** *NUN*, *NNNV*, *NNWV*, *NTUN*, *NTNN*,
5. **Errors and bad practices:** *NIERR*, *NDK*, *NBEH*, *NDM*, *NSCM*,
6. **Other complexity:** *NBL*, *NCD*, *NCO*, *NICD*, *NLP*, *NFL*, *NLO*, *NCMD*.

For each metric, we estimate a factor of generalizability (FoG), which is either *General*, *Semi-Specific*, or *Specific*. We consider a metric to be *General* if it is applicable to other languages without effort, *Semi-Specific* if it is only applicable to Ansible in the current form, but can be translated to different IaC languages, and *Specific* if it is only applicable to Ansible. A full overview of the metrics and their self-explanatory nature along with their FoG score are recapped in Table 2. We discuss these metrics briefly in the following sections.

3.4.1 | General complexity metrics

General complexity metrics are metrics that have little relation to Ansible code specifically. They can be created for other languages with ease, and therefore, their FoG is *General*. General complexity metrics tell something about the general size and structure of the script.

Every script has a number of *lines of code* (`LOC`). It was one of the first software development quality metrics, and there is no reason not to take it into account. Together with LOC, there's the number of *blank lines of code* (`BLOC`), and the relative number of `BLOC`: `BLOC/LOC`. We also count the number of *commented lines of code* (`CLOC`), as well as its relative counterpart: `CLOC/LOC`. We also count the number of *tokens* (`NTKN`) in a script which represents an interesting dimension: it cannot be semi-specific since some infrastructure languages are not consistent with tokenization, making it difficult to predict their properties and complexity.

The last metrics in this category are the *depth* (`DPT`) of a playbook, the number of *keys* (`NKEYS`) in a playbook, and its *entropy* (`ETP`). Because Ansible playbooks are written as `YAML`-code, they can be depicted as dictionaries. The depth of such a dictionary, is what we call the depth of the playbook. The same goes for the number of keys in such a dictionary.

3.4.2 | Task and module related metrics

According to the documentation, *each playbook is composed of one or multiple plays, and the goal of a play is to map a group of hosts to well-defined roles, represented by tasks.*[35] A play usually consists of executing tasks. A task itself can be seen as a call to an Ansible module. *The goal of each task is to execute a module with very specific arguments.*[35] Because other languages like Chef and TOSCA also allow to use and execute modules, but using a different syntax, the FoG of these metrics is *Semi-Specific*.

To start, we count the number of *tasks* (`NTS`) in a playbook. A larger task size, denotes more modules to be executed, denoting a more complex playbook. Therefore, we define the *average task size* (`ATSS`) as $\frac{1}{NTS}\sum_{i=1}^{NTS} TS_i$. Where TS_i the size of task *i*.

Each task is able to use and create variables. By the Ansible documentation, these are called *task variables*. We count the number of task variables (`NTVR`), the number of modules (`NMD`) and the number of *fact* modules (`NFMD`). A fact module is is defined as a module that does not alter state, but rather returns data (facts) about the playbook.

3.4.3 | External metrics

An Ansible playbook uses a lot of external information by *including* and *importing* external sources. Importing external sources is something that is used in every programming language, but the different types of import and the syntax differs. Therefore, the FoG of these metrics is *Semi-Specific*.

We count external metrics by counting the number of *include* (`NINC`), *include_role* (`NINCR`), *include_task* (`NINCT`), and *include_vars* (`NINCV`) statements. In the same way, we count the number of *import_playbook* (`NIMPP`), *import_role* (`NIMPR`), and *import_task* (`NIMPT`) statements. Last, we count the number of *lookup* commands (`NLK`), the number of *shell* commands (`NSH`), and the number of *external* modules (`NEMD`).

3.4.4 | Naming conventions

In Ansible, everything can be named. The best practices page of the Ansible documentation recommends to always name tasks, to use variables for naming them, and to use unique names.[35,36]

Therefore, we count the number of *unique names* (`NUN`), the number of names with a variable (`NNWV`) and with no variable (`NNNV`), the number of tasks with a unique name (`NTUN`), and the number of tasks with no name (`NTNN`).

Puppet, for example, does not allow for unnamed tasks.[37] Therefore, the FoG of NTNN is *Specific*, for the rest of the above metrics, the FoG is *Semi-Specific*.

3.4.5 | Errors and bad practices

Ansible offers the possibility to ignore errors with the *ignore_errors* command. Keywords and modules can be used, despite being deprecated. *Blocks* allow for errors to be handled, similar to exceptions in other programming languages.[38] We count the number of *ignore_errors* (`NIERR`), the number of *deprecated keywords* (`NDK`) and *modules* (`NDM`), and the number of blocks used for error handling (`NBEH`).

Last, we count the number of *suspicious comments* (`NSCM`). A comment is considered suspicious when it contains any of the following keywords: `TODO, FIXME, HACK, XXX, CHECKME, DOCME, TESTME, PENDING`.

In Puppet, there is no way to ignore errors. Therefore, the FoG of NIERR is *Semi-Specific*. Every language contains (different) deprecated keywords and modules, and offers methods for error handling, giving an FoG of *Semi-Specific* for NDK, NDM, and NBEH. Comments are language independent, giving an FoG of *General* to NSCM.

3.4.6 | Other complexity metrics

These are the metrics that are quite regularly used for different programming languages, which give some indication on the complexity of a program but that are not as straightforward to compute as the general complexity metrics, and not as easy to generalize for different languages due to different syntax. The FoG of all metrics in this category is *Semi-Specific*.

The metrics contained in this category are as follows: The number of *conditionals* (`NCD`), *inline conditionals* (`NICD`), *comparison operands* (`NCO`), *loops* (`NLP`), *filters* (`NFL`), *logic operands* (`NLO`), *command* commands (`NCMD`), and the number of *blocks* (`NBL`).

3.5 | Data preprocessing

As suggested by Aggarwal,[39] we use an independent ensemble framework of outlier detection techniques to classify outliers in our dataset, since independent ensembles are popularly used with high dimensional data.[40] The idea behind independent ensembles of outlier detection techniques is that different algorithms are applied to either the complete data or portions of the data. The results of these algorithms can then be combined to obtain more robust outliers.[39] We use a total of nine different outlier detection algorithms, as proposed by Zhao et al.[40] The techniques used can be found in Table 3. As suggested by Aggarwal[39] and Zhao et al,[40] we use the maximization method for score combination. Since outliers can be very relevant for the purpose of our research, we construct two additional types of fluid datasets: With and without outliers.

TABLE 3 Outlier detection techniques

Abbreviation	Technique
AKNN	Average *k* nearest neighbors
CBLOF	Cluster-based local outlier factor
FB	Feature bagging
HBOS	Histogram based outlier detection
IF	Isolation forest
KNN	*k* nearest neighbors
LOF	Local outlier factor
MCD	Minimum covariance determinant
OCSVM	One class support vector machine

3.6 | Clustering

In order to get more insight in our dataset, we divide the data in clusters and use multiple classification tools to find the metrics that are most influential to these clusters. As suggested by Domingos,[41] we use an ensemble clustering method for our cluster analysis. A clustering ensemble framework consists of three main components: Ensemble member generation, a consensus function, and evaluation.[42] The empirical study of Fern et al[43] on cluster ensembles on high dimensional data suggests using a random projection (RP)-based approach with spectral graph partitioning[44] as a consensus function. We use Silhouette (SH), Davies Bouldin (DB), and Calinski Harabasz (CH) scores as our evaluation metrics.

The resulting clusters are used as target variables for three classification algorithms: Random Forest, Extra Trees (ET), and XGBoost (XGB) classification. By doing so, we can extract the features that are most relevant for the clusters found in the previous step.

In the following subsections we elaborate on the clustering ensemble method and the parameters chosen for our model.

3.6.1 | Random projection based ensembles

Let $f:\mathbb{R}^{n\times d}\rightarrow\mathbb{R}^{d\times d'}$ a linear transformation given by $f(X)=X\cdot R=X'$, where $R\in\mathbb{R}^{d\times d\prime}$ a random projection matrix with $r_{ij}\sim N\left(0,\frac{1}{d'}\right)\forall i\in d,\forall j\in d'$ i.i.d. The resulting matrix X' is clustered following a k-means algorithm to create $k\in\mathbb{N}$ clusters.

This process is repeated $m\in\mathbb{N}$ times to create m clusterings $c_1,\ldots,c_m$, each containing k clusters.

3.6.2 | Spectral graph partitioning

The points in our dataset and the clusterings from the previous part, can be represented as a bipartite graph $G(U,V,E)$ so that each vertex $u\in U$ represents a cluster $u\in C_i$ $(i\in 1,\ldots,m$ and each vertex $v\in V$ represents an instance in our dataset. This means that for each edge $e=(e_1,e_2)\in E$ holds that either $e_1\in U$ and $e_2\in V$ or $e_1\in V$ and $e_2\in U$. Let A the connectivity matrix between points in U and V. We can write $G(U,V,E)$ as $W=\begin{bmatrix}0 & A^T\\ A & 0\end{bmatrix}$.[43] We use W to calculate the normalized Laplacian matrix L and create a matrix A consisting of the eigenvectors belonging to the k greatest eigenvalues of L. By using k-means to cluster the points in A, we assign each point from our dataset to one of k clusters.

3.6.3 | Parameter selection

According to Fern et al.,[43] d' needs to be chosen so that at least 90% of the explained variance of the data is kept. Therefore, we choose $d'=2$. n is the number of rows in the used dataset, and d the number of columns. We let k go from 2 to 10 clusters and use the SH, DB, and CH scores to determine the optimal number of clusters.

3.7 | Prediction modelling and model evaluation

To test our findings, we create a prediction model that predicts whether a playbook contains bugs or not. We use the results found in the clustering procedure to label the data, where playbooks in the deviating cluster are considered as "buggy" and playbooks in the other clusters are considered sound.

To create a test set, we used the retrieved GitHub repositories. By looking in the projects for commits that closed a "bug"-labeled issue, we were able to retrieve files consisting of bugs. The resulting files of these commits were labeled sound.

Following Rahman and Williams[6] who used a random forest classifier earlier for IaC scripts, we trained several classifiers: Random Forest, Decision Tree, Naive Bayes, and a Multilayer Perceptron. Each classifier is trained on each dataset, both on all the metrics, as well as on the most important metrics, found during the clustering procedure, with the deviating cluster as target variable.

4 | DESCRIPTIVE STATISTICS

4.1 | Metrics overview

The 428 Ansible repositories consist of 9290 Ansible files. We find no suspicious comments, no math operands, and no imported playbooks in our entire data set. There are 5 rows in the data set containing NaN values. After deletion of these metrics and rows, we are left with a Table of 9285 rows and 50 columns. The data has a sparsity of $\frac{\#zeros}{\#rows \times \#columns} \approx 0.62$. Meaning that, although over half of the data being 0, the data are not very sparse. A correlation plot of the most highly correlated variables can be found in Figure 14. In the following subsections, we discuss each metric shortly.

4.1.1 | General complexity metrics

The value distribution of the metrics can be found in Figures 3 and 4. To keep the visualization insightful, the data are cut off at the .95th percentile. We note that LOC, NTKN, and NKEYS are relatively equally distributed. Most playbooks have depth 1,2, or 3. CLOC becomes constant, and BLOC is almost constant. This means that playbooks for which $BLOC > 0$ or $CLOC > 0$ might be interesting for our prediction. As expected, LOC, NTKN and NKEYS share strong correlations. ETP has a negative correlation with LOC, NTKN, DPT, and NKEYS, indicating a lower entropy for larger, more complex playbooks. Relative counts share a correlation with their absolute counterparts.

4.1.2 | Task and module related metrics

The value distribution of the metrics can be found in Figures 5 and 6. The data are cut off at the .95th percentile. NFMD becomes constant with this cutoff. NTS, NTVR, and NMD seem to be quite equally distributed. As seen in Figure 6, they are correlated as well. For most of the playbooks,

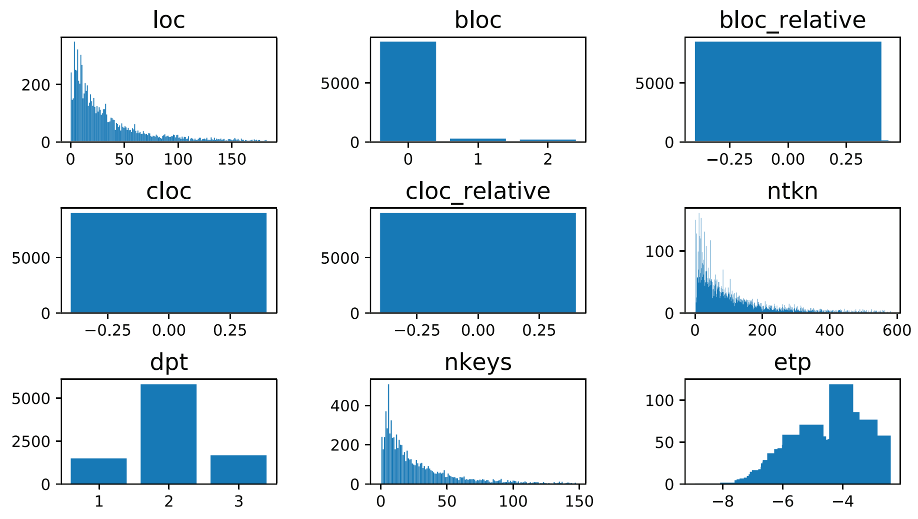

FIGURE 3 Counts of general complexity metrics

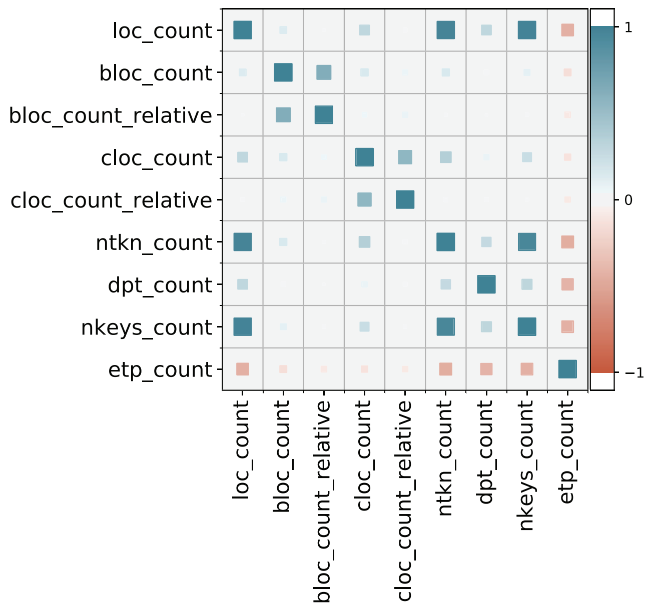

FIGURE 4 Correlations of general complexity metrics

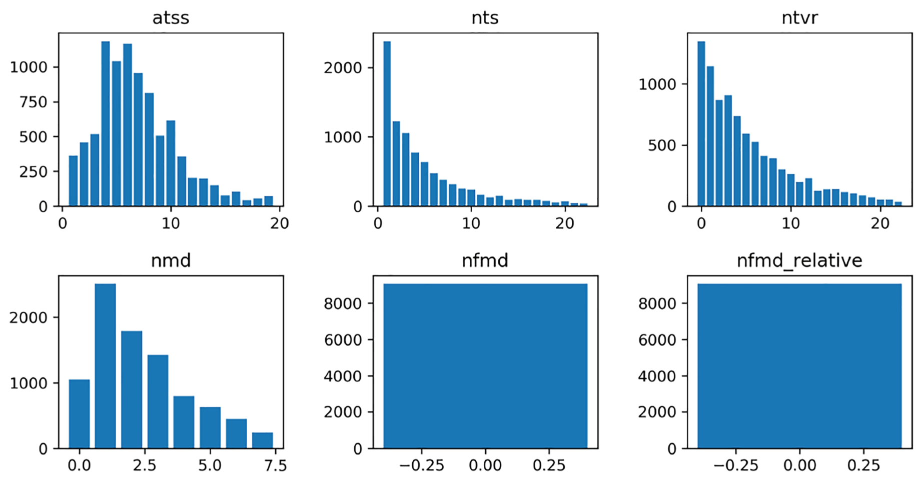

FIGURE 5 Counts of task and module related metrics

their values are 0 or close to 0. ATSS is generally greater than 0, but smaller than 10. We conclude that for the largest part, task and module related metrics are not too high.

4.1.3 | External metrics

For visualization purposes, we omit figures and representations concerning this dimension as We note that very little playbooks get information from external sources. Also, no correlation is found between the variables involved in this dimension.

4.1.4 | Metrics related to naming

The value distribution of the metrics can be found in Figure 7. The data are cut off at the .95th percentile. NUN, NNNV, and NTUN seem to be fairly evenly distributed. As can be seen in Figure 8, unnamed tasks (NTNN) show a general negative correlation with other name giving metrics. Surprisingly, we see no correlation between the relative variables and their absolute counterparts. We conclude again that most of these variables are generally low valued.

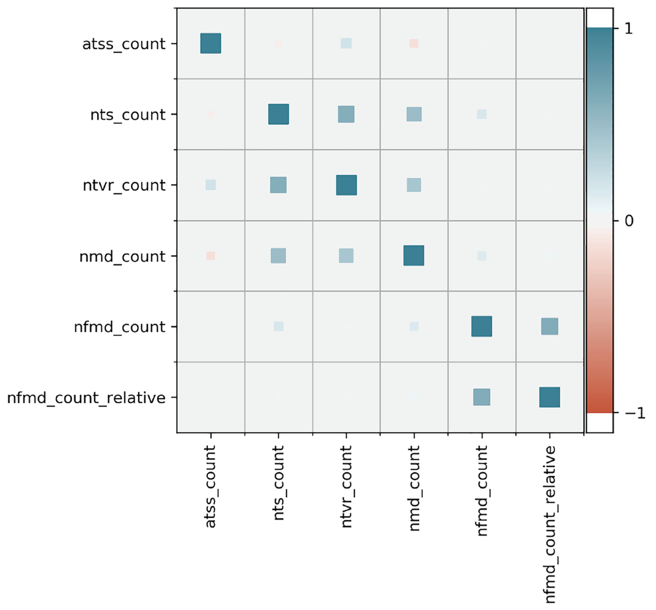

FIGURE 6 Correlations of task and module related metrics

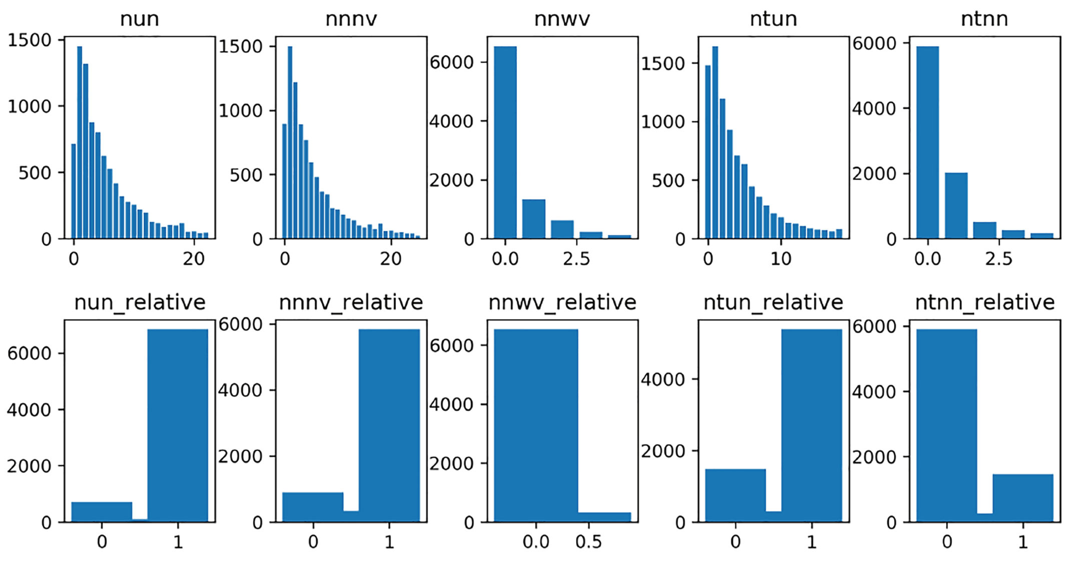

FIGURE 7 Counts of naming metrics

4.1.5 | Complexity metrics

The value distribution of the metrics can be found in Figure 9. The data are cut off at the .95th percentile. We notice that conditionals tend to occur most in the playbooks, whereas loops occur the least amount of times. However, most of the playbooks do not report having any of them. All the variables are slightly correlated, as can be seen in Figure 10.

4.1.6 | Errors and bad practices

The correlations between the metrics can be found in Figure 11. There are correlations between each absolute metric and its *occurrences* counterpart. The playbooks with a relatively high number for these metrics were considered interesting for our analysis as they reflect considerable conditions according to the infrastructure code programming model.

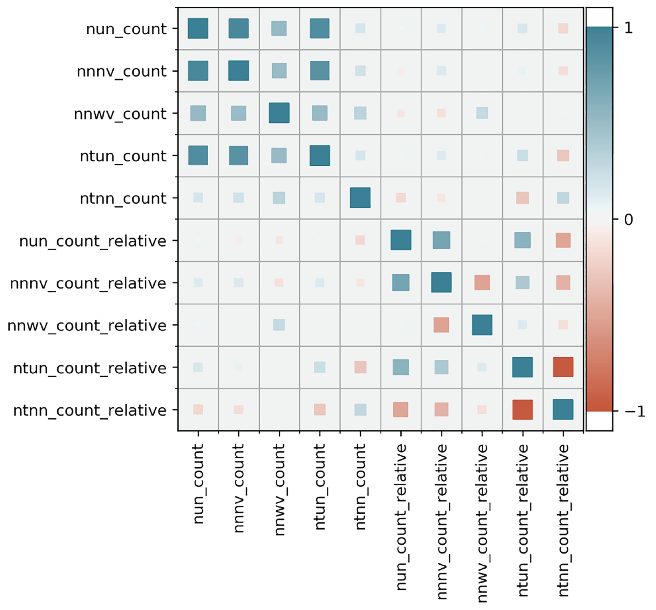

FIGURE 8 Correlations of naming metrics

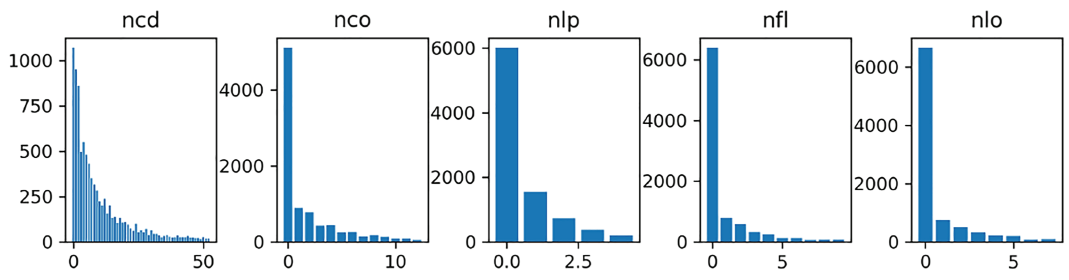

FIGURE 9 Counts of other complexity metrics

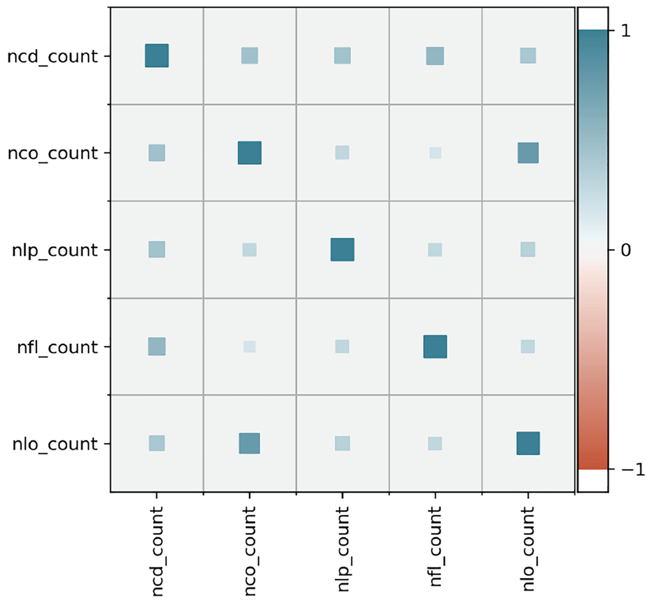

FIGURE 10 Correlations of other complexity metrics

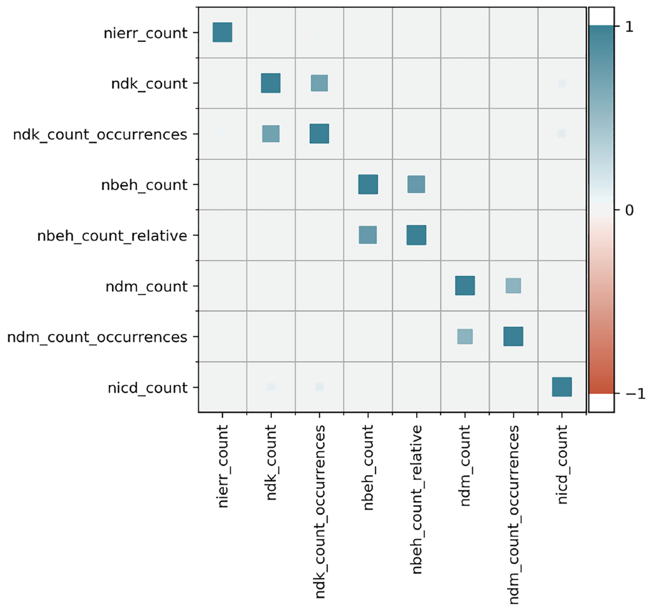

FIGURE 11 Correlations of error distributions

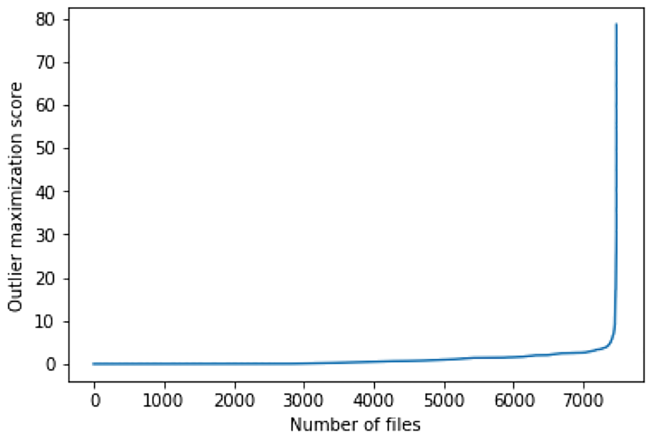

FIGURE 12 Outlier maximization score

4.2 | Outlier detection

We find 397 outliers, which is roughly 4.27% of our data. The behavior of the outlier maximization score (OMS) can be found in Figure 12. When comparing the behavior of outliers and inliers, we find that for the metrics NDM, NDM_occurrences, NFMD, NFMD_relative, NIMPR, NIMPT, NINC, NINCR, NINCT, NINCV, the distribution within the set of outliers is not significantly different from the distribution within the set of inliers. In Table 4, we see that outliers generally have a lower value than inliers for the metrics: ETP, NDK_Occurrences, NNNV_relative, NTUN_relative, and NUN_relative. We expected NDK_occurrences to have higher values in the set of outliers. Generally, in the inliers, more variables are used in task names, probably meaning that more variables are created dynamically. We also find more unique names in the playbooks belonging to the set of inliers. In general, the comparison of the outlier distribution with the inlier distribution satisfies our expectations: outliers tend to have higher values, indicating more complex playbooks.

5 | CLUSTERING RESULTS

As discussed in Section 3.2, we create four different datasets. Each dataset is considered with and without outliers. We refer to the resulting datasets with the following abbreviations:

TABLE 4 Significance of each inlier (most significant in red color) vs. outlier (most significant in blue color) distribution

Metric	Avg inliers	Avg outliers	Significant	*p* value
atss	8.65	47.48	True	5.90e−21
bloc	0.27	3.09	True	3.20e−11
bloc_relative	0.01	0.03	True	5.76e−7
cloc	0.05	0.61	True	1.49e−4
cloc_relative	0.0	0.0	True	3.06e−2
dpt	2.06	2.79	True	1.85e−34
etp	−4.91	−5.85	True	2.36e−67
loc	41.31	265.64	True	7.85e−118
nbeh	0.02	0.27	True	1.27e−10
nbeh_relative	0.02	0.18	True	1.27e−10
nbl	0.17	0.88	True	1.24e−16
ncd	11.49	69.35	True	2.00e−99
ncmd	0.34	0.97	True	9.90e−5
nco	2.32	10.22	True	1.16e−16
ndk	0.09	0.26	True	3.12e−10
ndk_occurrences	0.13	0.8	True	3.12e−10
ndm	0.0	0.03	False	1.00e+0
ndm_occurrences	0.0	0.11	False	1.00e+0
nemd	1.03	4.36	True	2.67e−38
nemd_relative	0.22	0.34	True	5.01e−14
nfl	1.72	9.79	True	8.02e−23
nfmd	0.04	0.3	False	2.69e−1
nfmd_relative	0.0	0.02	False	2.69e−1
nicd	0.19	1.41	True	2.20e−5
nierr	0.2	1.43	True	2.45e−6
nimpr	0.02	0.07	False	1.00e+0
nimpt	0.12	0.39	False	5.62e−1
ninc	0.18	0.89	False	9.99e−1
nincr	0.03	0.08	False	1.00e+0
ninct	0.26	0.68	False	9.03e−1
nincv	0.04	0.1	False	9.88e−1
nkeys	34.11	212.08	True	2.01e−112
nlk	0.03	0.49	True	3.31e−5
nlo	1.16	5.84	True	3.74e−10
nlp	0.78	3.18	True	1.87e−17
nmd	2.61	4.29	True	2.39e−16
nnnv	6.06	33.29	True	2.26e−94
nnnv_relative	0.84	0.71	True	1.87e−47
nnwv	0.68	7.63	True	3.87e−52
nnwv_relative	0.09	0.24	True	3.87e−52
nsh	0.31	1.1	True	5.92e−3
ntkn	132.25	851.33	True	8.85e−124
ntnn	0.81	3.34	True	5.25e−12
ntnn_relative	0.23	0.28	True	5.25e−12
nts	5.77	26.6	True	3.83e−64
ntun	4.64	19.63	True	2.62e−59

TABLE 4 (Continued)

Metric	Avg inliers	Avg outliers	Significant	*p* value
ntun_relative	0.75	0.64	True	2.90e−25
ntvr	5.91	22.15	True	1.13e−66
nun	5.8	27.35	True	2.13e−101
nun_relative	0.87	0.74	True	8.44e−51

TABLE 5 Cluster sizes per data set

Data set	c0	c1	c2	c3	c4	c5
AM	6440	1697	487	174	335	152
AMNO	339	1038	2281	484	4548	198
ABS	6434	1546	418	237	509	141
ABSNO	2252	4524	1037	203	487	385
REL	6464	1551	395	475	155	245
RELNO	2253	4560	1040	517	348	170
NOCOR	6268	1724	651	165	400	77
NOCORNO	1100	4418	2185	460	489	236

Note: The colored cells represent the suspicious clusters.

TABLE 6 Most important features per table

AM	AMNO	ABS	ABSNO	REL	RELNO	NOCORR	NOCORRNO
ntkn	ntkn	ntkn	ntkn	ntkn	ntkn	loc	loc
loc	loc	loc	loc	loc	loc	atss	atss
nkeys	nkeys	nkeys	nkeys	nkeys	nkeys	etp	etp
ncd	ntun	ntun	etp	ncd	etp	nmd	nfl
etp	nun	ncd	nun	nts	ncd	nfl	nlo
nnnv	ncd	etp	nts	etp	atss	nemd_rel	nmd
nun	etp	atss	ncd	ntvr	nts	nlo	nlp
nts	nmd	nts	atss	atss	ntvr	ntnn	ntnn
ntun	atss	nun	ntun	nmd	nmd	nlp	nemd_rel
nlp	ntvr	nnnv	ntvr	nlp	nsh	nun_rel	nun_rel

1. AM: a dataset containing all metrics,
2. AMNO: AM without outliers,
3. ABS: a dataset containing only absolute metrics,
4. ABSNO: ABS without outliers,
5. REL: a dataset where we keep only the relative counterpart if an absolute metric has one,
6. RELNO: REL without outliers,
7. NOCOR: a dataset without all the highly correlated metrics,
8. NOCORNO: NOCOR without outliers.

For each dataset, we compute 2 − 10 clusters and evaluate the results using the Silhouette (SH) score, Davies Bouldin (DB) score, and the Calinski Harabasz (CH) score. A high SH and CH score, and a low DB score are indications of good clusterings. For each dataset, we see that the optimal number of clusters equals 6. For each dataset, the size of each cluster can be found in Table 5

We explore the resulting clusters by using different classification methods to identify the metrics most relevant to the clusters. By looking at the results in Table 6, we see that the influence of correlated metrics is heavily present in the AM, ABS, and REL tables, with NTKN, LOC, NKEYS, NCD, NTS, and NUN being important metrics. For each dataset, they are distributed according to the figures in the following subsections. For

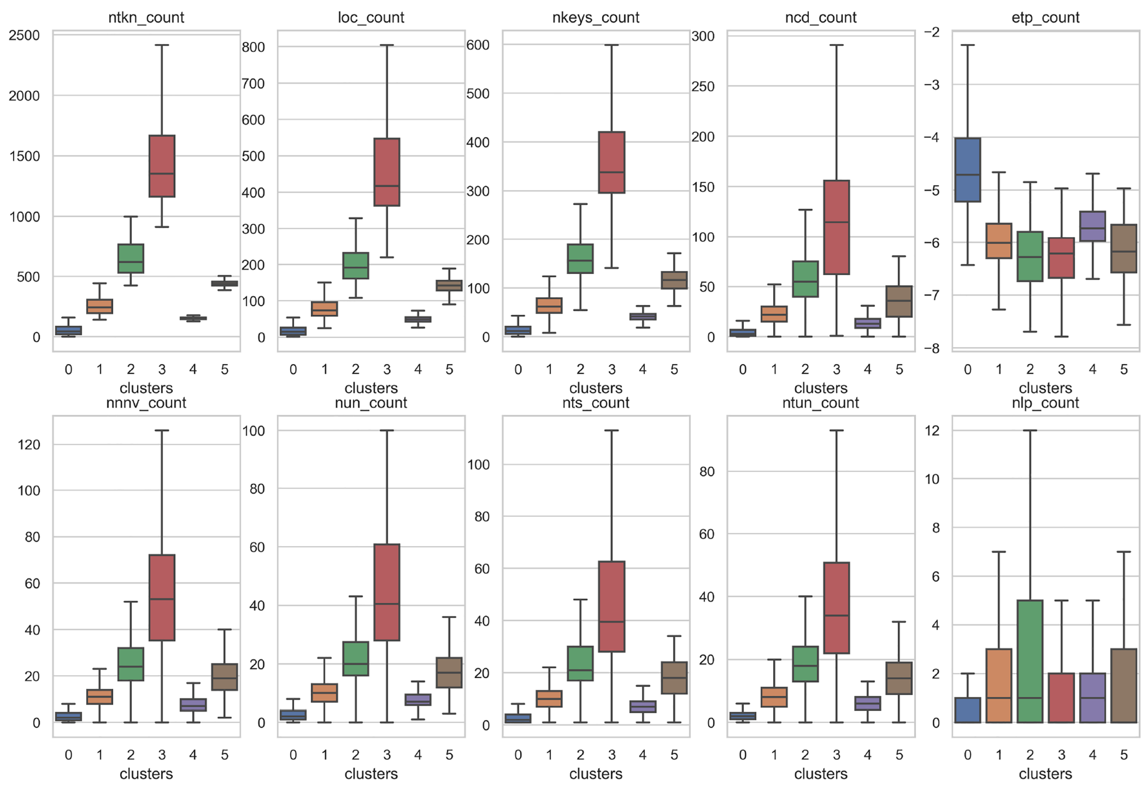

FIGURE 13 Distribution of the most important metrics in the AM dataset

every dataset, we note one cluster behaving very different from the other datasets: a Suspicious Cluster. For each dataset *A* in the set of datasets, we define the Suspicious Cluster to be SC_A. In the following sections we will elaborate on the contents of this cluster and whether we expect to find defects. The distribution of important metrics for the AM dataset can be found in Figure 13.

5.1 | All metrics (AM)

In Figure 13, we see the distribution of the most important metrics per cluster for the AM dataset. We immediately note cluster 3 as behaving very different from the other clusters: for 8 out of 10 metrics, the values in the cluster are much higher than for the other clusters. The values in the cluster are also more widely distributed than for the other clusters. This could also be attributed to the small size of cluster 3 (with 174 playbooks), however, note that cluster 5 also contains only 152 playbooks. We do note in Figure 14 that NTKN, LOC, NCD, NKEYS, NNNV, NTS, NTUN, and NUN are highly correlated metrics, which can also be seen in their behavior in the boxplots. Note that the median number of loops equals 0 for clusters 0, 3, and 5.

For the Playbooks in SC_{AM}, this means that they are generally much larger playbooks: more tokens, lines of code, conditionals, and variables. More *stuff* happens in these playbooks, suggesting that there is also a higher chance of things going wrong.

5.1.1 | No outliers (AMNO)

For a large part, this dataset holds the same metrics as most important as the AM dataset. The metrics NNNV, NTS, and NLP are replaced for NMD, ATSS, and NTVR. The suspicious cluster in this dataset is cluster 5, consisting of 198 Playbooks. Note that for 53 playbooks $p \in SC_{AMNO}$ holds that $p \in SC_{AM}$.

In comparison to the other clusters, the playbooks in SC_{AMNO} are larger. Just like for the AM dataset. However, we note that outliers are removed from this dataset: where the median number of tokens for AM lies around 1400, for AMNO, this value lies just above 800.

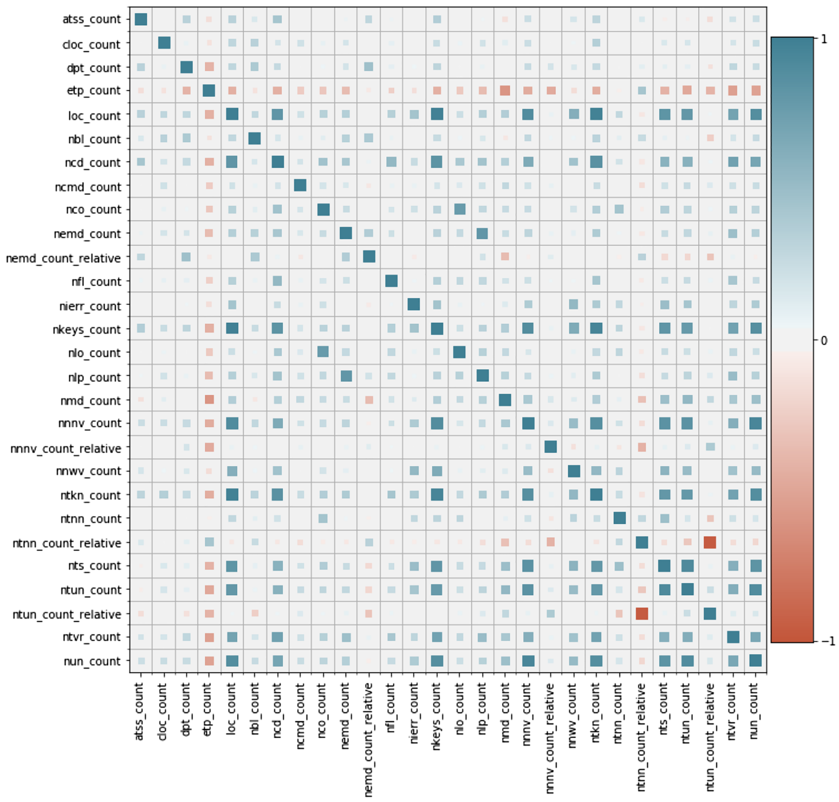

FIGURE 14 Correlation plot of most highly correlated metrics

5.2 | Absolute metrics (ABS)

We immediately note one suspicious cluster, just like in the previous datasets. This dataset introduces no new metrics. SC_{ABS} consists of 141 Playbooks. For 40 Playbooks $p \in SC_{ABS}$ holds $p \in SC_{AMNO}$. For all $p \in SC_{ABS}$ holds $P \in SC_{AM}$.

5.2.1 | No outliers (ABSNO)

We find cluster 3 to be of interest, consisting of 205 playbooks. For 38 playbooks $p \in SC_{ABSNO}$, we find that $p \in SC_{ABS}$. For 191 playbooks $p \in SC_{ABSNO}$, we find that $p \in SC_{AMNO}$.

5.3 | Relative metrics (REL)

We note that cluster 4 sticks out. No new metrics are introduced. For 154 playbooks $p \in SC_{REL}$ holds $p \in SC_{AM}$. For 39 playbooks $p \in SC_{REL}$, we find $p \in SC_{AMNO}$.

5.3.1 | No outliers (RELNO)

For this dataset, cluster 5 sticks out. Less important metrics seem more evenly distributed than the more important metrics. NSH is newly introduced as an important metric, for which the .75th quartile is equal to 1 in cluster 2 and 3. However, due to its deviation compared to other clusters in the more important metrics, we remain more interested in cluster 5 than in clusters 2 and 3. For 18 playbooks $p \in SC_{RELNO}$, we find $p \in SC_{AM}$. For 163 playbooks $p \in SC_{RELNO}$, we find $p \in SC_{AMNO}$.

5.4 | No Correlated metrics (NOCOR)

We see cluster 5 deviating the most for the most important metrics. In this dataset, we notice a real difference from the previously investigated. Because the highly correlated metrics are dropped, a lot of new metrics are introduced. However, for a lot of metrics, the behavior of the suspicious cluster does not deviate very much from the other metrics. However, all the outlying ATSS values seem to be included in the suspicious cluster. This cluster also seems to contain playbooks with, in general, a lower amount of unique names (NUN_relative). We find that for 64 playbooks $p \in SC_{NOCOR}$ holds $p \in SC_{AM}$. For 9 playbooks, $p \in SC_{NOCOR}$ holds $p \in SC_{AMNO}$.

5.4.1 | No outliers (NOCORNO)

Interesting is that where the number of filters behaved differently for a different cluster in the previously discussed dataset, for this dataset, the number of filters stands out in cluster 5, our suspicious cluster. We find that for 51 playbooks $p \in SC_{NOCORNO}$ holds $p \in SC_{AM}$. For 182 playbooks, $p \in SC_{NOCORNO}$ holds $p \in SC_{AMNO}$.

5.4.2 | Suspicious clusters

We see that the number of tokens, lines of code, and the number of keys are the most important metrics for all datasets, except NOCOR and NOCORNO. For NOCOR(NO), where NTKN, and NKEYS (amongst others) are removed, the number of lines of code is the most important metric. The entropy of a playbook, and the average task size are also reoccurring metrics. Despite the entropy being an important metric for our classifiers, it doesn't seem to be very differently distributed from the other metrics in the suspicious clusters, in contrast to the other metrics. NTKN, LOC, and NKEYS always seem to be much higher for the playbooks in the suspicious clusters.

For the playbooks in the suspicious cluster, this means that they are generally larger and more complex than the other playbooks: They consist of more lines of code and more conditionals. We also see a higher *average* task size for each SC, meaning that also on a deeper level we see the author trying to do more in a single step. In Table 5, we see the size of each Suspicious cluster. Note that for every dataset A, except for AM SC_A is also the smallest cluster in the dataset. This could explain the high variability within each cluster. Were it not that c5 from AM does not share the high variability of SC_{AM} and that c3 from ABS is about the same size as some of the SCs, without sharing their variability.

In Table 7, we see the number of playbooks present in suspicious clusters belonging to different datasets. We note that the SCs of datasets containing outliers have a lot in common, as well as the SCs of datasets containing no outliers.

Our findings here seem to be in agreement with,[17] who conclude that the reliability of a program decreases once the complexity increases. Our playbooks that deviate most from the other playbooks have a higher complexity, which suggests that their reliability might be lower than that of other playbooks.

6 | DEFECT PREDICTION MODELLING

6.1 | Using the suspicious cluster

We have trained three classification models (Random Forest, Decision Tree, Naive Bayes) to predict defects in Ansible Playbooks in eight different datasets. We trained each classifier two times for each dataset: Once using all the features in the dataset, and once using only the identified most

TABLE 7 The number of playbooks found in the suspicious cluster of both tables

	AM	AMNO	ABS	ABSNO	REL	RELNO	NOCORR
AM	174						
AMNO	51	198					
ABS	141	36	141				
ABSNO	51	197	36	203			
REL	155	40	141	40	155		
RELNO	18	163	3	169	7 170		
NOCORR	64	9	62	9	63	3	77
NOCORRNO	51	182	36	183	40	150	9

TABLE 8 Accuracy (acc), precision (prec), recall (rec), and F1 score of predictions using the Suspicious cluster as a target variable

	clf	all acc	all prec	all rec	all F1 score	imp acc	imp prec	imp rec	imp F1 score
AM metrics	RF	0.51	0.50	0.07	0.12	0.51	0.50	0.07	0.12
	DT	0.51	0.50	0.07	0.12	0.51	0.50	0.07	0.12
	NB	0.51	0.50	0.23	0.32	0.51	0.50	0.10	0.17
AMNO	RF	0.49	0.40	0.07	0.11	0.51	0.50	0.10	0.17
	DT	0.51	0.50	0.10	0.17	0.51	0.50	0.10	0.17
	NB	0.51	0.50	0.23	0.32	0.51	0.50	0.27	0.35
ABS	RF	0.51	0.50	0.07	0.12	0.51	0.50	0.07	0.12
	DT	0.51	0.50	0.07	0.12	0.51	0.50	0.07	0.12
	NB	0.52	0.53	0.27	0.36	0.49	0.40	0.07	0.11
ABSNO	RF	0.51	0.50	0.10	0.17	0.51	0.50	0.10	0.17
	DT	0.51	0.50	0.10	0.17	0.51	0.50	0.10	0.17
	NB	0.51	0.50	0.33	0.40	0.51	0.50	0.27	0.35
REL	RF	0.51	0.50	0.07	0.12	0.51	0.50	0.07	0.12
	DT	0.51	0.50	0.07	0.12	0.51	0.50	0.07	0.12
	NB	0.52	0.53	0.30	0.38	0.51	0.50	0.17	0.25
RELNO	RF	0.51	0.50	0.03	0.06	0.51	0.50	0.03	0.06
	DT	0.51	0.50	0.03	0.06	0.51	0.50	0.03	0.06
	NB	0.51	0.50	0.23	0.32	0.51	0.50	0.27	0.35
NOCORR	RF	0.51	0.50	0.03	0.06	0.51	0.50	0.03	0.06
	DT	0.51	0.50	0.03	0.06	0.51	0.50	0.03	0.06
	NB	0.52	0.56	0.17	0.26	0.51	0.50	0.10	0.17
NOCORRNO	RF	0.51	0.50	0.07	0.12	0.51	0.50	0.07	0.12
	DT	0.49	0.40	0.07	0.11	0.51	0.50	0.07	0.12
	NB	0.52	0.53	0.30	0.38	0.51	0.50	0.33	0.40

Note: We use "all" to refer to predictions using all metrics in a data set, and "imp" to refer to predictions using only the important metrics.

TABLE 9 Accuracy (acc), precision (prec), recall (rec), and f score of predictions using Outliers as a target variable

		all acc	all prec	all recall	all fscore	imp acc	imp prec	imp recall	imp fscore
AM	RF	0.52	0.57	0.13	0.22	0.51	0.50	0.10	0.17
	DT	0.51	0.50	0.20	0.29	0.49	0.44	0.13	0.21
	NB	0.49	0.48	0.43	0.46	0.49	0.44	0.13	0.21
ABS	RF	0.52	0.57	0.13	0.22	0.49	0.40	0.07	0.11
	DT	0.52	0.57	0.13	0.22	0.48	0.43	0.20	0.27
	NB	0.52	0.52	0.43	0.47	0.51	0.50	0.10	0.17
REL	RF	0.49	0.40	0.07	0.11	0.51	0.50	0.10	0.17
	DT	0.51	0.50	0.10	0.17	0.46	0.36	0.13	0.20
	NB	0.51	0.50	0.43	0.46	0.51	0.50	0.20	0.29
NOCORR	RF	0.52	0.60	0.10	0.17	0.51	0.50	0.07	0.12
	DT	0.52	0.60	0.10	0.17	0.48	0.40	0.13	0.20
	NB	0.48	0.47	0.57	0.52	0.51	0.50	0.33	0.40

Note: We use "all" to refer to predictions using all metrics in a data set, and "imp" to refer to predictions using only the important metrics.

important features in the dataset. The results can be found in Table 8. Predictions using all metrics are referred to as *all*, predictions using only important metrics are referred to as *imp*. We calculate the accuracy (acc), precision (prec), and recall (rec).

We find that each classifier gets about the same accuracy and precision. However, for the recall and the F1 score, we do find significant differences. The Naive–Bayes classifier consistently gives the best result for almost each dataset. The highest F1 score is obtained for the ABSNO and the NOCORNO datasets for all the metrics and only the important metrics respectively.

6.2 | Modelling using outliers

The defect prediction scores using outliers as classification target can be found in Table 9. Using the files we classified as outliers, we find an *f* score of 0.52 for the Naive–Bayes classifier using only uncorrelated metrics on all data. This is a slight improvement compared to classifying based on the suspicious clusters. This increase in *f* score is mainly due to an improvement of the recall, meaning that fewer playbooks have incorrectly been classified as sound. However, using outliers as a target variable seems to make the overall classification worse. Using only the important metrics, NB is much less consistent.

6.3 | Bug detection based on code changes

An interesting aspect of our test set is that it consists of the same files, before and after debugging. In Table 10, we can see the number of files that are correctly predicted as defect before the commit and correctly predicted as sound after the commit, trained on the suspicious clusters. Table 11 shows the same results based on outliers.

Using suspicious clusters, we see that the Naive–Bayes classifier is most consistent and most successful in predicting bugs based on changes in code, with a best accuracy score of 0.23.

TABLE 10 Number of correctly identified bug fixing commits using clusters as target variable

		Using all metrics	Using important metrics
AM	RF	0	0
	DT	0	0
	NB	1	1
AMNO	RF	0	1
	DT	1	1
	NB	0	3
ABS	RF	0	0
	DT	0	0
	NB	7	0
ABSNO	RF	1	1
	DT	1	1
	NB	1	3
REL	RF	0	0
	DT	0	0
	NB	3	3
RELNO	RF	1	1
	DT	1	1
	NB	0	3
UNCORR	RF	1	1
	DT	1	1
	NB	4	1
UNCORRNO	RF	0	0
	DT	0	0
	NB	4	3

TABLE 11 Number of correctly identified bug fixing commits using outliers as target variable

		Using all metrics	Using important metrics
AM	RF	2	1
	DT	1	2
	NB	2	1
ABS	RF	2	0
	DT	2	2
	NB	2	2
REL	RF	0	1
	DT	1	2
	NB	2	3
UNCORR	RF	1	1
	DT	1	0
	NB	2	3

6.4 | Research solution

As a result of this study, we design an Ansible defect prediction tool. We create a tool that reads an Ansible file, as selected by the user, extracts all metrics as mentioned in this report, and returns a prediction on whether the Ansible file is sound or not, using the Naive Bayes predictor that returned the best result in our research. The prediction model can be found on GitHub.[¶] Furthermore, to encourage verifyability, all materials and scripts needed for the experimentation reported in this paper are available online.[#]

7 | DISCUSSION AND FUTURE RESEARCH

7.1 | Findings and outlook

From a software engineering perspective, the lack of technical debt management approaches to 'quantifying' infrastructure code makes it relatively impossible to predict its qualities. At the same time, while several works are emerging to address such shortcoming, there is still a need to look for more complex and effective measurements that cope with infrastructure code variety and its relatively wide technical space.

In the scope of the experimentation in this manuscript, we attempted to identify defectiveness as a property of generalisable metrics, changing the shape of the data being fed to a predictive modelling approach.

As a result of our experimentation, there are at least three evident limitations of this emerging field, stemming from this research: (1) There is a need for interpretative layers on top of the predictions to make such predictions actionable—in our case, we observed that even though there are models capable of yielding some useful predictions, their actionability is limited if not accompanied by an assessment of the individual metric impact; (2) defective does not mean erroneous—although our predictive modelling indicates reasonable defectiveness areas, runtime defect prediction approaches would be able to use such information pro-actively, and perhaps more formal specification of runtime checking for infrastructure code may be able to avoid such risks during orchestration. These mechanisms are not a reality yet and must be explored; (3) in the scope of our work, modelling multiple infrastructure languages and analysing them at the same time and with the same predictive modelling approach turned out to be more difficult than it originally seemed as, for example, modelling formats do not share the same programming model and it is therefore difficult to prepare predictive analytics accounting for their collective peculiarities—consequently, abstraction-based modelling of infrastructure code and model-checking of its properties during design time is both still in its infancy and useful, for example, to represent the analyzed DevOps pipelines for their troubleshooting.

7.2 | Research goals and addressing research questions

The goal of our study was to develop traditional software development quality metrics in IaC scripts in order to predict the quality of said scripts. We find that metrics related to the size of a script give the best quality indications. We also find results that are an improvement of the current state of the art, which is provided by the SZZ algorithm of Borg et al,[31] who found an F1 score of $[0.10 - 0.15]$, a recall of $[0.13 - 0.20]$, and a precision of $[0.07 - 0.12]$.

To answer **RQ1**, we use the results of our defect prediction in Section 3.7. On a test set consisting for 50% of sound files and 50% of defect files, our best result is an F1 score of .52, with a recall of .57 and a precision of .47. This is a significant improvement compared to Borg et al,[31] who found an F1 score of $.10 - .15$, a recall of $.13 - .2$, and a precision of $.07 - .12$, which is the current state of the art. The improvement on the recall score compared to the current state of the art means that fewer scripts have incorrectly passed the test as *sound*.

To answer **RQ2**, we have tried to correctly classify bug-fixing commits on Ansible scripts as such. Using Random Forest and Decision Tree classifiers, our best score is an accuracy of .03 in 16 different settings. Using a Naive Bayes classifier, in seven out of 16 settings, we find an accuracy smaller than .03 when recognizing bug fixing commits. In eight out of 16 settings, we find an accuracy of $[.10 - .13]$, and in one setting we find an accuracy of .23. By being able to outperform RF and DT, Naive Bayes provides an interesting lead for identifying bug fixing commits.

To answer **RQ3**, we have revised the total set of 50 different metrics part of our fluid dataset creation exercise. A subset of these metrics is derived from traditional software development quality metrics, making them easily generalizable for different IaC languages. The largest part of the set of metrics is designed specifically for Ansible scripts, but are measurements of properties found in many different programming languages, as discussed in Section 4.1. These metrics can be calculated for different IaC languages, but a different syntax has to be taken into account.

The most important metrics for identifying and predicting quality of IaC scripts can be found in Table 6. The most important metrics that are easily generalizable for other IaC languages are: `ETP, LOC, NCD, NKEYS, NLO, NSH, NTKN, NUN, NUN_REL`. The metrics that can be used in other IaC languages, but require a bit more work to retrieve are: `ATSS, NEMD_REL, NFL, NLP, NMD`. The important metrics specific for Ansible are: `NNNV, NTNN, NTUN, NTS, NTVR`. This means that of the 19 metrics that help identifying defects in IaC scripts, 9 are very easily transferable to other languages, 5 are transferable with a manual work and extensions to our framework,[45] and 5 metrics are specific for Ansible.

Because[29] have used smells in Puppet code to identify smells in Chef, we expect that with some extra effort, identifying and predicting the quality of IaC scripts of other languages, such as Chef, Puppet, and TOSCA, can be done with the same metrics we used for Ansible.

It must be noted that the current implementation of our tool will not flag the source code as defected if a proper correction is applied. In addition to that, the model can be trained with user-feedback (e.g., users that mark an artifact as not-defected) making its precision improve during utilization (active-learning).

7.3 | Limitations and future research

The reliability of the results is limited by the size and the selection method of our test data. The test data consists of only 61 files, while our models are trained on 9290 files, giving an imbalance in the split of our train and test data. Our study also operates under the assumption that a commit containing the keyword `FIXED` fixes a bug. While we can be quite certain that the script before the commit contains (a) bug(s), we cannot be sure that the script after the commit is bug free.

The results we have found are promising, but future research following up on our results should build a bigger test dataset. By taking a closer look into automatically labeled data, or by manually labeling data using experts, more reliable predictions can be made. We recommend to do more research on the parameter settings of the Naive Bayes classifier and to develop tests to check our conclusions on the generalizability of our results for other IaC languages.

Moreover, we plan to employ quality metrics that are specific to IaC (e.g., the ones presented in Dalla Palma et al.[11]) in addition to general ones as we did in this work. A proper comparison will not only potentially improves the results obtained herein but also helps understanding the need and the effectivness of ad hoc metrics.

Finally, our future work includes exploring the addition of an explainability layer that will enable users to not only detect defected code but also have hints on the reasons why the problem arose.

8 | CONCLUSION

By using traditional software development quality metrics, we have succeeded to make an improvement on the current state of the art in predicting bugs in an IaC script. By developing these metrics using existing literature and the Ansible documentation, we created several versions of the same potential dataset, leading to the possibility to make clusters of our data set; with this approach, we were able to find a subset of Ansible scripts showing a different behavior than the other scripts. We have shown that these clusters can be used quite well to find bugs in Ansible scripts, and that they provide an interesting lead for recognizing bug fixing commits.

By trying to predict defects in Ansible scripts, we have provided a follow up on the research of Rahman et al[6] and Borg et al.[31] We also made a contribution to the existing gap of studies on IaC defects, as mentioned by related works by Rahman et al.[5]

Because of the use of simple, measurable metrics, we expect it possible to improve on our results by creating a larger and more thoroughly inspected test data set, and by constructing and deriving more complex metrics, like McCabe's cyclomatic complexity metric.[46]

At the same time, in the scope of our research design, we chose to focus our efforts around the underlying assumption that a IaC commit with the FIXED keyword does actually fix a bug since informal focus groups that drove the inception of our research design made us confident that the aforementioned decisions/assumption actually reflects the state of industrial practice. Nevertheless, we recon that a focused study on how infrastructure code is actually patched and how bugs are triaged in its scope should be warranted with further study.

An important part of the metrics we have constructed can be generalized for other IaC languages, like Chef, Puppet, and TOSCA. By creating a model independent of the language that is able to extract said metrics and make a prediction on the quality of the script, it will be easier to find defects before bringing the script to production, independent of the used IaC language, thereby reducing the chance for infrastructure downtime.

ACKNOWLEDGMENTS

The authors would like to thank Mr. Kasper Koojiman for his invaluable work in preparing several of the main portions of this work as well as editing several portions of the manuscript. Open Access Funding provided by Politecnico di Milano within the CRUI-CARE Agreement.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in GitHub at https://github.com/kckooijman/iac-defect-predictor/tree/master/Data.

ORCID

Giovanni Quattrocchi https://orcid.org/0000-0002-0405-9814
Damian Andrew Tamburri https://orcid.org/0000-0003-1230-8961

ENDNOTES

[*] This assumption was worked out by confirming our research design within a focus group held in conjunction with the OASIS TOSCA—which stands for Topology and Orchestration for Cloud Applications—Standardization group.

[†] https://www.ansible.com.

[‡] https://www.jetbrains.com/lp/devecosystem-2019/devops.

[§] https://jenkins.io/doc/.

[¶] https://github.com/kckooijman/ansible-defect-predictor.

[#] https://github.com/kckooijman/ansible-metrics-defect-prediction.

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; 2017:497-498.
3. Gartner. The Cost of Downtime. 2014. https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/
4. Morris K. *Infrastructure as Code: Managing Servers in the Cloud*: " O'Reilly Media, Inc."; 2016.
5. Rahman A, Mahdavi-Hezaveh R, Williams L. Where are the gaps? A systematic mapping study of infrastructure as code research. arXiv preprint arXiv: 180704872; 2018.
6. 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; 2018:34-45.
7. Khan K, Ratner E, Ludwig R, Lendasse A. Feature bagging and extreme learning machines: Machine learning with severe memory constraints. In: IJCNN. IEEE; 2020:1-7. http://dblp.uni-trier.de/db/conf/ijcnn/ijcnn2020.html#KhanRLL20
8. Ng V, Cardie C. Bootstrapping coreference classifiers with multiple machine learning algorithms. In: EMNLP; 2003. http://dblp.uni-trier.de/db/conf/emnlp/emnlp2003.html#NgC03
9. Fulcher J. Computational intelligence: An introduction. In: Fulcher J, Jain LC, eds. *Computational Intelligence: A Compendium*, Studies in Computational Intelligence, vol. 115: Springer; 2008:3-78. http://dblp.uni-trier.de/db/series/sci/sci115.html#Fulcher08
10. Palma SD, Nucci DD, Palomba F, Tamburri DA. Towards a catalogue of software quality metrics for infrastructure code. 2020. CoRR abs/2005.13474. http://dblp.uni-trier.de/db/journals/corr/corr2005.html#abs-2005-13474
11. Dalla Palma S, Di Nucci D, Palomba F, Tamburri DA. Toward a catalog of software quality metrics for infrastructure code. *J Syst Softw*. 2020;170: 110726.
12. Li HF, Cheung WK. An empirical study of software metrics. *IEEE Trans Softw Eng*. 1987;6:697-708.
13. Tang M-H, Kao M-H, Chen M-H. An empirical study on object-oriented metrics. In: Proceedings Sixth International Software Metrics Symposium (Cat. No. pr00403) IEEE; 1999:242-249.
14. Xenos M, Stavrinoudis D, Zikouli K, Christodoulakis D. Object-oriented metrics-a survey. In: Proceedings of the FESMA; 2000:1-10.
15. Yu S, Zhou S. A survey on metric of software complexity. In: 2010 2nd IEEE International Conference on Information Management and Engineering IEEE; 2010:352-356.

16. Keshavarz G, Modiri N, Pedram M. Metric for early measurement of software complexity. *Interfaces*. 2011;5(10):15.
17. Rashid J, Mahmood T, Nisar MW. A study on software metrics and its impact on software quality. arXiv preprint arXiv:190512922; 2019.
18. Song Q, Jia Z, Shepperd M, Ying S, Liu J. A general software defect-proneness prediction framework. *IEEE Trans Softw Eng*. 2010;37(3):356-370.
19. Yu X, Keung J, Xiao Y, Feng S, Li F, Dai H. Predicting the precise number of software defects: are we there yet? *Inf Softw Technol*. 2022;2022:106847.
20. Rajnish K, Bhattacharjee V, Gupta M. A novel convolutional neural network model to predict software defects. In: Fundamentals and Methods of Machine and Deep Learning: Algorithms, Tools and Applications; 2022:211-235.
21. Tong H, Wang S, Li G. Credibility based imbalance boosting method for software defect proneness prediction. *Appl Sci*. 2020;10(22):8059.
22. Humphreys J, Dam HK. An explainable deep model for defect prediction. IEEE / ACM; 2019:49-55. http://dblp.uni-trier.de/db/conf/icse/raise2019.html#HumphreysD19
23. Nam J. Survey on software defect prediction. *Tech. Rep*, Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology; 2014.
24. Wan Z, Xia X, Hassan AE, Lo D, Yin J, Yang X. Perceptions, expectations, and challenges in defect prediction. *IEEE Trans Softw Eng*. 2018;46:1241-1266.
25. Vendome C, Linares-Vásquez M, Bavota G, Di Penta M, German D, Poshyvanyk D. License usage and changes: a large-scale study of java projects on github. In: Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension IEEE Press; 2015:218-228.
26. Reyes López A. Analyzing github as a collaborative software development platform: A systematic review; 2017.
27. Horton E, Parnin C. Gistable: Evaluating the executability of python code snippets on Github. In: 2018 ieee international conference on software maintenance and evolution (icsme) IEEE; 2018:217-227.
28. Schermann G, Zumberi S, Cito J. Structured information on state and evolution of dockerfiles on Github. In: Proceedings of the 15th international conference on mining software repositories ACM; 2018:26-29.
29. Schwarz J, Steffens A, Lichter H. Code smells in infrastructure as code. In: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC) IEEE; 2018:220-228.
30. Rahman A. Characteristics of defective infrastructure as code scripts in DevOps. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings ACM; 2018:476-479.
31. Borg M, Svensson O, Berg K, Hansson D. Szz unleashed: an open implementation of the SZZ algorithm—featuring example usage in a study of just-in-time bug prediction for the Jenkins project. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation - MaLTeSQuE 2019; 2019. https://doi.org/10.1145/3340482.3342742
32. Hammouri A, Hammad M, Alnabhan M, Alsarayrah F. Software bug prediction using machine learning approach. *IJACSA) Int J Adv Comput Sci Appl*. 2018;9(2):78-83.
33. Garriga. Mining GitHub Microservices. 2017. https://github.com/webocs/mining-github-microservices
34. Red Hat I. Ansible Playbook Documentation. 2019. https://docs.ansible.com/ansible/latest/user_guide/playbooks.html
35. Red Hat I. Ansible Playbook Introduction. 2019. https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html
36. Red Hat I. Ansible Best Practices Documentation. 2019. https://docs.ansible.com/ansible/latest/user_guide/playbooks_best_practices.html
37. Documentation P. Puppet Tasks Documentation. 2018. https://puppet.com/docs/pe/2018.1/writing_tasks.html
38. Red Hat I. Ansible Error Handling Documentation. 2019. https://docs.ansible.com/ansible/latest/user_guide/playbooks_error_handling.html
39. Aggarwal CC. Outlier ensembles: position paper. *ACM SIGKDD Explor Newslett*. 2013;14(2):49-58.
40. Zhao Y, Nasrullah Z, Li Z. Pyod: A python toolbox for scalable outlier detection. *J Mach Learn Res*. 2019;20(96):1-7. http://jmlr.org/papers/v20/19-011.html
41. Domingos P. A few useful things to know about machine learning; 2012.
42. Alqurashi T, Wang W. Clustering ensemble method. *Int J Mach Learn Cybern*. 2019;10(6):1227-1246.
43. Fern XZ, Brodley CE, et al. Cluster ensembles for high dimensional clustering: An empirical study; 2006.
44. Ng AY, Jordan MI, Weiss Y. On spectral clustering: Analysis and an algorithm. In: Advances in neural information processing systems; 2002:849-856.
45. Dalla Palma S, Di Nucci D, Tamburri DA. Ansiblemetrics: a python library for measuring infrastructure-as-code blueprints in ansible. *SoftwareX*. 2020;12:100633.
46. McCabe TJ. A complexity measure. *IEEE Trans Softw Eng*. 1976;4:308-320.

How to cite this article: Quattrocchi G, Tamburri DA. Predictive maintenance of infrastructure code using "fluid" datasets: An exploratory study on Ansible defect proneness. *J Softw Evol Proc*. 2022;34(11):e2480. doi:10.1002/smr.2480