Infrastructure-as-Code Defect Prediction Using Program Dependence Graph Metrics

Valeria Pontillo*, Gerardo Iuliano*, Ruben Opdebeeck†,
Coen De Roover†, Dario Di Nucci*, Fabio Palomba*, Filomena Ferrucci*
vpontillo@unisa.it, g.iuliano29@studenti.unisa.it, ruben.denzel.opdebeeck@vub.be,
coen.de.roover@vub.be, ddinucci@unisa.it, fpalomba@unisa.it, fferrucci@unisa.it
*Software Engineering (SeSa) Lab — University of Salerno, Fisciano, Italy
†Software Languages (Soft) Lab — Vrije Universiteit Brussel, Belgium

***Abstract*—Context: Infrastructure-as-code (IaC) is a DevOps practice that facilitates the management and provisioning of infrastructure by utilizing machine-readable files known as IaC scripts. Similarly to other types of source code artifacts, these scripts are susceptible to defects that may hinder their functionality. Objective: We conjecture that Program Dependence Graph (PDG) metrics may provide insights into the defectiveness of IaC scripts and, based on such a conjecture, we propose to develop and empirically evaluate a new defect prediction model based on PDG metrics. Method: We plan to extract the PDG metrics from 137 open-source Ansible projects and train five machine learners to assess their capabilities in a within-project scenario, other than comparing them with a state-of-the-art defect predictor relying on structural and process IaC-oriented metrics. Finally, we plan to assess the performance of a combined model that mixes together PDG and existing IaC-oriented metrics.**

***Index Terms*—Infrastructure-as-Code; Defect prediction; Software Quality; Empirical Software Engineering.**

I. Introduction

DevOps [17] enables the automation of the software lifecycle at both development and operation levels. In this context, *Infrastructure as Code* (IaC) has emerged as the practice of automating the process of deploying and maintaining infrastructure through executable source code scripts [28]. Configuration management tools, e.g., Ansible [22], Chef [18], and Puppet [27], enable practitioners to employ IaC scripts for configuring the machines in their infrastructure programmatically. For instance, they can install software dependencies and manage configuration files. The adoption of these tools has been increasing rapidly [21], with Ansible emerging as one of the most popular solutions [25]. Nonetheless, Infrastructure as Code remains source code that may exhibit the same flaws as application code, e.g., code smells [31], [42]–[44], defects [35], [36], [40], [41], or security concerns [38]. These issues may notably impact the reliability and maintainability of infrastructure code and have several negative implications, e.g., hot service stand-by [3] or costly elastic provisioning [9].

> 👁 *In the context of our work, we focus on identifying defective Ansible scripts, attempting to extend the current state of the art related to using machine learning algorithms for defect prediction.*

More specifically, defect prediction [45] relies on machine learning algorithms to identify the portions of source code more likely to exhibit defects, allowing developers to take appropriate mitigation plans, e.g., test case prioritization. When considered in the context of IaC, the effective prediction of defect-prone IaC scripts may help organizations embracing the DevOps principles to focus on the most critical scripts during quality assurance activities and find a better way to allocate effort and resources. The current state of the art in IaC defect prediction is represented by the work by Dalla Palma et al. [14], who proposed the so-called RADON framework: this is a machine learning-based framework that supports the prediction of defect-prone Ansible scripts through the use of a mixture of product and process metrics, e.g., lines of code or average task size of an Ansible script. While the empirical study conducted on RADON showed that it may predict defect-prone scripts with high accuracy, Dalla Palma et al. [14] also pointed out that further metrics able to characterize orthogonal properties of Ansible scripts may further improve defect prediction capabilities.

> 🎯 *We* hypothesize *that novel metrics computed on top of a Program Dependence Graph (PDG) representation of Ansible scripts [31] may improve the performance of IaC defect prediction models.*

A PDG representation captures both the control and data flow of Ansible scripts, providing an improved mechanism to analyze the inner working of those scripts. This paper overviews our research method to address our hypothesis: we propose a *confirmatory empirical investigation* where we will (1) collect a new set of 17 metrics based on the program dependence graph of Ansible scripts—the definition of these metrics comes from the adaptation of concepts and metrics proposed by previous research; (2) devise a new defect prediction model based on the newly defined metrics; and (3) evaluate the performance of the model, assessing the contribution of the metrics both in isolation and when combined with the metrics employed by Dalla Palma et al. [14]. We finally assess statistically whether PDG metrics contribute to the prediction of defective IaC scripts.

II. Background and Motivation of Our Study

This section first provides a gentle introduction to Infrastructure as Code and, afterwards, presents a running example showing the rationale behind our hypothesis.

A. Infrastructure as Code

Infrastructure as code (IaC) is a method to create and manage computing environments where software systems will operate and be controlled using reusable scripts of infrastructure code [28]. There are many languages and platforms for different aspects of infrastructure management, such as tools for creating and managing virtual machines (Cloudify, Terraform), tools for handling container technologies (Docker Swarm, Kubernetes), tools for building machine images (Packer), and tools for configuring systems (Ansible, Chef, Puppet).

Our experiments focus on Ansible because it is one of the most popular tools among practitioners [21]. This automation engine leverages the YAML language and automates cloud provisioning, configuration management, and application deployment. In Ansible, a *playbook* defines an IT infrastructure automation workflow as a sequence of ordered *tasks* that apply to one or more *inventories* of managed infrastructure nodes. A *module* is a segment of code that a task invokes. A module has a specific purpose, for example, creating a MySQL database and installing an Apache webserver. A *role* combines tasks and resources that accomplish a specific goal, such as installing and configuring MySQL.

Listing 1 depicts an example of an Ansible playbook that deploys an application on all hosts in an inventory, as indicated on line 2. The playbook defines one variable, `app_version` (line 4), whose value is generated using the current timestamp. Subsequently, it defines two tasks, the first of which (lines 6–9) will ensure that a directory exists for this version using the `file` module, which manages file system contents. It refers to the `app_version` variable by means of an expression (line 8), demarcated by double braces. The second task (lines 10–13) then pulls sources from a git repository into this directory by referring to the same variable.

```
1  - name: Deploy application
2    hosts: all
3    vars:
4      app_version: "{{ lookup('pipe', 'date +%Y%m%d
       %H%M%S') }}"
5    tasks:
6      - name: Create directory
7        file:
8          path: "/app/releases/{{ app_version }}"
9          state: directory
10     - name: Pull sources into directory
11       git:
12         repo: https://github.com/my/repo
13         dest: "/app/releases/{{ app_version }}"
```

Listing 1. Example of an Ansible playbook.

B. Our Working Hypothesis Explained

Although Ansible code is written in YAML and, therefore, easy to parse, the resulting tree-based representation contains little information regarding the behavior of the script. For instance, the tree cannot link a variable reference in an expression to the definition of this variable (i.e., definition-use pairs). To alleviate this limitation, Opdebeeck et al. [31] proposed the Program Dependence Graph (PDG) representation for Ansible scripts. Their PDG is a graph capturing the control flow and data flow of an Ansible script. Nodes in the graph represent script elements, such as tasks, expressions, variables, and literal data values. Edges represent either control flow (between tasks) or various types of data flow (between data and tasks). To exemplify, Figure 1 depicts the PDG for the example provided in Listing 1. We can see the two tasks as oval nodes labeled by the module they invoke. Moreover, the expressions, literal values, and variables present in the playbook are depicted and connected via data-flow edges. Importantly, we can observe that there exist two separate nodes to represent the `app_version` variable. This behavior is because Ansible lazily evaluates variable initializers. Therefore, the initializer will be re-evaluated for each variable reference. Since the expression generating the current timestamp may change arbitrarily, each variable reference thus resolves to a different possible abstract value. In fact, *this is a defect in the example playbook*, which will create one directory but pull the git repository into another one. Based on this example, we can draw a key consideration: this defect may be solely identified using data flow analysis, as it is connected to the way variables are *used* rather than how they are *defined*. Consequently, metrics computed on top of a tree-based representation of Ansible scripts could not identify the defect. For this reason, Radon would fail for this case [14]. In contrast, a PDG representation may enable an orthogonal analysis of Ansible defects—our work defines novel metrics accounting for the usage of variables (e.g., the `edgesCount` later described in Table I). Therefore, we hypothesize that features extracted

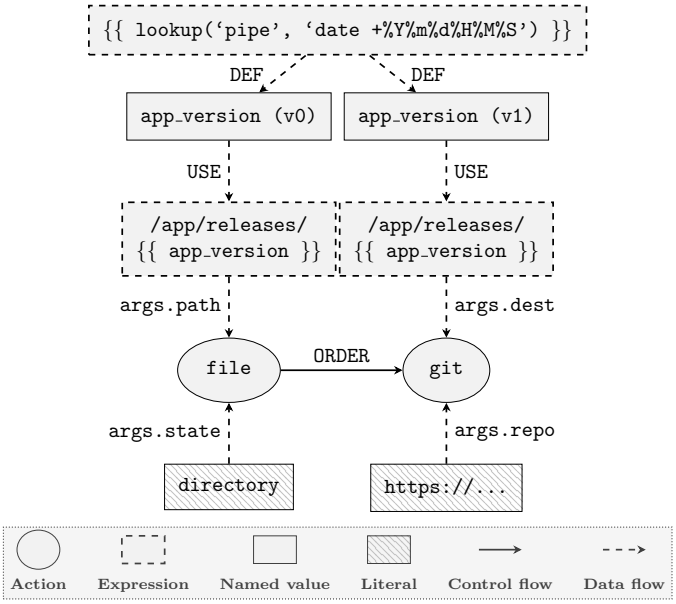

Fig. 1. Program dependence graph for the example of Listing 1.

from these graphs may enable machine learning models to predict defects more accurately. The remainder of the paper describes how this hypothesis will be addressed.

III. Empirical Study Design

The *goal* of the study is to evaluate whether metrics extracted from the program dependence graph are suitable for the defect prediction model in a within-project setup, with the *purpose* of improving the early detection of defects in IaC scripts. The *perspective* is of researchers who are interested in improving the effectiveness of defect prediction models applied in the context of Infrastructure as Code.

A. Research Questions

Our empirical investigation will aim at addressing the following research questions (**RQ**s):

RQ$_1$. *Which metrics related to the program dependence graph are good defect predictors?*

RQ$_2$. *What is the best defect prediction model based on the metrics derived from PDG?*

RQ$_3$. *To what extent the PDG model is complementary to the state-of-the-art model?*

RQ$_4$. *Does a combination of PDG-based, structural, and process metrics boost the performance of IaC defect prediction?*

With **RQ$_1$**, we seek to understand which metrics related to the program dependence graph contribute the most to detecting defects in IaC scripts. These observations will be used to (i) quantify the predictive power of metrics based on the program dependence graph (PDG) and (ii) identify the most promising features to include in a prediction model of failure-prone IaC scripts. In **RQ$_2$**, we employ the most promising metrics coming from **RQ$_1$** in experimentation aimed at establishing the best machine learning model relying on PDG metrics. We will then perform a further step ahead, namely that of understanding how complementary is the model coming from **RQ$_2$** with respect to the state-of-the-art model proposed by Dalla Palma et al. [14]. The outcome of **RQ$_3$** will reveal insights into the potential added value of the model based on PDG metrics: such potential will be finally quantified in **RQ$_4$**, where we will experiment with how different feature sets behave independently from each other and how they augment each other. As a last step, we will address our working hypothesis by defining a more specific pair of null and alternative hypotheses, whose validity will be statistically addressed. To design and report our study, we will follow the empirical software engineering guidelines by Wohlin et al. [47], other than the *ACM/SIGSOFT Empirical Standards*.[1]

[1] Available at https://github.com/acmsigsoft/EmpiricalStandards.

B. Context Selection

We will first collect data from the dataset of 137 open-source Ansible projects, publicly available on GitHub and released by Dalla Palma et al. [14]. The dataset will allow us to compare the performance of models trained using structural, process, and PDG metrics. As the number of defects in these projects ranges from nine to 1,658; we are likely to employ data balancing before training the within-project models.

C. Empirical Study Variables

The first step to answer the research questions posed in our study concerned the definition of the empirical study variables, namely (1) the dependent variable to predict and (2) the features to be used as independent variables.

1) Dependent Variable: The goal of our study is to automatically detect the presence of a defect in infrastructural code components. Therefore, as a dependent variable, we will rely on a binary value indicating the presence/absence of a bug.

2) Independent Variables: We have already performed a literature analysis to understand how PDG metrics could be applied in IaC. We obtained a set of metrics based on the analysis of program dependence graph [2], [34], which consider several characteristics such as the program size, complexity, coupling, and cohesion to capture the program behavior. However, such metrics were proposed for procedural code (i.e., implemented for the C language); therefore, we performed an additional analysis to tailor their concepts, e.g., slices and files, to IaC, e.g., tasks and playbooks. Table I shows the complete set of metrics we will experiment with. The metrics will be extracted using the PDGs proposed by Opdebeeck et al. [31], [32]. In this respect, it is worth noting that while we plan to operationalize all the metrics, we might still end up with inconsistency issues due to the assumptions made by PDGs proposed by Opdebeeck et al. [31], [32], e.g., *taskCount*, *taskCoverage*, *taskSize*, and *taskIdentifier* represent concepts close to some ICO metrics, whereas *pdgVerticesSum* and *taskSpatial* have a similar definition to *verticesCount* and *taskCoverage*. Should this happen, we might be enforced to exclude some metrics when executing our research plan.

Specifically, we will use their builder to create the PDG for each playbook and role in each project. However, their approach constructs whole-project PDGs that may represent multiple files, whereas our dependent variable has file-level granularity. Note that we cannot build a PDG for single files as this may incorrectly approximate data flow, as pointed out by Opdebeeck et al. [31]. Instead, we will implement a PDG slicer where each slice represents an individual file. We can then extract the metrics listed in Table I for each individual slice. To validate the implemented PDG slicer and metrics extractor, we will perform a manual analysis. We plan on selecting a statistically significant sample of the PDG slices (confidence level=95%, margin of error=5%) and, for each slice, we will manually compare the slice to the file from which it originates to validate the slicer. Moreover, we will manually calculate the PDG metrics for the sample to validate our metric extractor and to gauge the accuracy of the metrics.

This validation will be carried out by the second author, who will go through the slices and assess whether they include all the information of the tasks. On a weekly basis, the first and third authors will virtually meet the second author to (i) double-check the activities performed and (ii) discuss corner cases or complex slices that could not be successfully analyzed by the second author alone. Upon execution of the study, we will discuss the major insights from the manual validation, reporting the number of times the slices will perfectly match the tasks, other than the outcome of the weekly meetings.

D. Machine Learning for Defect Prediction

The following illustrates how we plan to leverage machine learning classification.

1) Selecting Machine Learning Algorithms: In the context of our study, we will experiment with multiple machine learning classifiers. First, we will include *Naive Bayes* [16] and *Logistic Regression* [26] as classifiers that do not require much training data. Then, we will consider *Decision Tree* [20], *Random Forest* [8], and *Support Vector Machine* [30], which are more flexible and powerful classifiers. The selection is mainly driven by our willingness to conduct a fair comparison with the state of the art. Indeed, our study builds on top of the findings by Dalla Palma et al. [14] and verifies the contributions brought by PDG metrics to IaC defect prediction: as such, we opt for the use of the same set of classifiers used in the baseline study [14]. In this way, our work may provide insights into the usefulness of PDG metrics as defect predictors by keeping the same working environment as Dalla Palma et al. [14] in an effort to provide the research community with results that may be more easily interpreted and compared. The assessment of more sophisticated approaches, e.g., deep learning, should therefore be considered out of the scope of this study and part of future research efforts.

2) Configuration and Training: When training the selected machine learners, we need to consider that class imbalance is one of the major obstacles to proper classification by supervised learning algorithms [4]. This observation is particularly true in defect prediction, where the neutral class outnumbers the failure-prone class. We will experiment with several under- and over-sampling configurations to overcome this obstacle. Specifically, we will consider using the NEARMISS 1, NEARMISS 2, and NEARMISS 3 algorithms for the under-sampling. Finally, we will experiment with a RANDOM UNDERSAMPLING approach that randomly explores the distribution of majority instances and under-samples them. As for the over-sampling, we will experiment with *Synthetic Minority Over-sampling Technique* (SMOTE) and advanced versions of this algorithm, i.e., *Adaptive Synthetic Sampling Approach* (ADASYN) and the BORDERLINE-SMOTE. We will also experiment with a RANDOM OVERSAMPLING approach that randomly explores the distribution of the minority class and over-samples them. Two main observations drive the selection of these balancing techniques. On the one hand, they make different assumptions on the underlying data distribution, hence allowing us to experiment with multiple algorithms and evaluate how the built prediction models react to them - the insights coming from this analysis would benefit the research community, which may learn more about how different data balancing solutions affect IaC defect prediction models. On the other hand, these techniques were also experimented with by Dalla Palma et al. [14]: as explained earlier, this choice allows us to compare our results with previous ones more fairly.

Fig. 2. Walk-forward validation process.

The training data will be normalized, scaling numeric attributes. We plan to evaluate three configurations for data normalization, namely (i) no normalization, (ii) *min-max* transformation to scale each feature individually in the range [0,1], and (iii) *standardization* of the features by removing the mean and scaling to unit variance.

Finally, we plan to configure the hyper-parameters of the selected machine learning classifiers by using the RANDOM SEARCH algorithm [5], which randomly samples the hyper-parameters space to find the best combination of hyper-parameters maximizing a scoring metric (in our case, the Matthews Correlation Coefficient). We plan to develop the entire pipeline with the SCIKIT-LEARN library in PYTHON.

3) Validation of the Approach: To assess the performance of our models, we will perform a within-project validation to understand how accurate the performance can be when a defect prediction model is trained using data from the same project where it should apply. The model selection will be guided by a randomized search of the models' parameters through a walk-forward validation [19]. In a walk-forward validation, the dataset represents a time series that can be divided into chronologically orderable parts, e.g., a project release. In each run, all data available before the part to predict will be used as the training set, while the part to predict will be used as the test set, preventing the test set has data antecedent to the training set. Afterward, the model performance will be computed as the average of various runs. Figure 2 shows the validation process. Specifically, the number of iterations will be equal to the number of parts minus one. We will train each model on the first n releases and test on the (n+1)-th release.

IV. EXECUTION PLAN

A. $\boldsymbol{RQ}_1$ - *In Search of Suitable Program Dependency Graph Metrics for Defect Prediction Models*

To evaluate the relative predictive power of the metrics described in Section III-C2, we will perform recursive feature selection to find the metrics that maximize the performance and rank them according to their importance for the prediction.

TABLE I
METRICS LEVERAGING PROGRAM DEPENDENCE GRAPHS IN FUNCTIONAL PROGRAMMING CONTEXTUALIZED TO INFRASTRUCTURE AS CODE

Reference	Original PDG Metric	Description	IaC-PDG Metric	Description
[2], [34]	sliceCount	Number of slices a file contains. $sliceCount(x) = k$, where k is the number of slices in file x.	taskCount	Number of tasks a playbook contains. $taskCount(x) = k$, where k is the number of tasks in playbook x.
[2]	sliceSize	Average number of lines of code (LOC) in a module's slices. $sliceSize(x) = \sum_{i=1}^{k} S_i/k$, where S_i is the number of LOC in slice i and k is the number of slices in module x.	taskSize	Average number of lines of code (LOC) in a playbook's tasks. $taskSize(x) = \sum_{i=1}^{k} S_i/k$, where S_i is the number of LOC in task i and k is the number of tasks in playbook x.
	sliceIdentifier	Average number of distinct occurrences of programmer-defined labels within a slice. $sliceIdentifier(x) = \sum_{i=1}^{k} SI_i/k$, where SI_i is the number of identifiers in slice i, and k is the number of slices in module x.	taskIdentifier	Average number of distinct occurrences of programmer-defined labels within a task. $taskIdentifier(x) = \sum_{i=1}^{k} SI_i/k$, where SI_i is the number of identifiers in task i, and k is the number of tasks in playbook x.
	sliceSpatial	Average spatial distance in LOC between the definition and the last use of the slice divided by the module size. $sliceSpatial(x) = \sum_{i=1}^{k} sliceDistance(i)/k$, where k is the number of slices in x. $sliceDistance(i) = (Sm_i - Sn_i)/q$, where Sm_i is the line number of the first statement in slice i, Sn_i is the line number of the last statement in slice i, and q is the module size in LOC.	taskSpatial	Average spatial distance in LOC between the definition and the last use of the task divided by the file size. $taskSpatial(x) = \sum_{i=1}^{k} taskDistance(i)/k$, where k is the number of tasks in x. $taskDistance(i) = (Sm_i - Sn_i)/q$, where Sm_i is the line number of the first statement in task i, Sn_i is the line number of the last statement in task i, and q is the playbook size in LOC.
	sliceCoverage	Average ratio between the slice sizes and the file's LOC.	taskCoverage	Average ration between the task's sizes and the playbook's LOC.
[34]	verticesCount	Number of vertices in a function's PDG.	verticesCount	The number of vertices in a task's PDG.
	edgesCount	Number of edges in a function's PDG	edgesCount	Number of edges in a task's PDG
	edgesToVerticesRatio	Ratio between the number of dependence edges and the number of vertices PDG for a given function's PDG.	edgesToVerticesRatio	Ratio between the number of dependence edges and the number of vertices PDG for a given task's PDG.
	sliceVerticesSum	Sum of the vertices contained in each function's slice.	pdgVerticesSum	Sum of the vertices contained in each playbook's task
	maxSliceVertices	Number of vertices of the slice's PDG with the maximum number of vertices in all function's slices of a module.	maxPdgVertices	Number of vertices of the task's PDGs with the maximum number of vertices in all task's PDGs of a playbook.
	globalInput	Number of parameters and non-local variables in a function.	globalInput	The number of parameters and non-local variables in a task.
	globalOutput	Number of non-local variables modified in a function.	globalOutput	Number of non-local variables modified in a task.
	directFanIn	Sum of the number of slices in other modules that use the output variables directly modified in a function.	directFanIn	Sum of the number of tasks in other playbooks that use the output variables directly modified in a task.
	indirectFanIn	Sum of the number of slices in other modules that use the output variables indirectly modified in a function.	indirectFanIn	Sum of the number of tasks in other playbooks that use the output variables indirectly modified in a task.
	directFanOut	Sum of the number slices in other modules whose output variables are directly modified and used in a function.	directFanOut	Sum of the number of tasks in other modules whose output variables are directly modified and used in a task.
	indirectFanOut	Sum of the number slices in other modules whose output variables are indirectly modified and used in a function.	indirectFanOut	Sum of the number of tasks in other modules whose output variables are indirectly modified and used in a task.
	lackOfCohesion	The number of shared vertices between function's slices.	lackOfCohesion	The number of shared vertices between playbook's tasks.

Given an external estimator that assigns weights to features, recursive feature elimination (RFE) will select features by recursively considering smaller and smaller sets of features that optimize the performance criteria. Specifically, the algorithm trains the estimator on the initial set of features and ranks the features by importance. The least important features are pruned from the current set. This procedure will be recursively repeated on the pruned set until the algorithm selects the desired number of features. Indeed, RFE requires selecting the number of features to keep, which is often unknown in advance. To find the optimal number of features, we will apply cross-validation to score the different feature subsets and select the best-scoring collection of features. To this end, we will employ the RFECV method[2] available in SCIKIT-LEARN.

B. $\boldsymbol{RQ}_2$ - *In Search of the Best Defect Prediction Model based on Program Dependency Graph Metrics*

To assess the performance of our models, we will experiment with multiple combinations, e.g., we will experiment with how the performance varies when including (and excluding) the normalization or the data balancing steps.

To address $\mathbf{RQ}_2$, we will first compute metrics such as *precision*, *recall*, *F-Measure*, *Matthews Correlation Coefficient* (*MCC*), and the *Area Under the Curve - Precision-Recall (*AUC-PR*)*. In addition, to account for the imbalanced nature of the dataset exploited, we will also compute micro and macro averages of the metrics, i.e., variants of the evaluation metrics that weight the performance achieved by a model according to the distribution of the two classes, i.e., defective and non-defective IaC scripts. Afterward, we will establish the best model by comparing the MCC of the experimented models through the Wilcoxon's rank test [46], applying the post-hoc Bonferroni's correction [29] to deal with the multiple comparisons that will be performed during the validation process. In addition, we will also compute the Cohen's δ effect size measure [12] to assess the magnitude of the differences observed. We will still compute and discuss the additional evaluation criteria metrics, i.e., *F-Measure*, *Precision*, *Recall*, and *AUC-PR*, to provide a more comprehensive overview of the capabilities of the experimented models. For these evaluation metrics, we will report statistics (i.e., mean, median, minimum, maximum, and standard deviation) about the classifier achieving the best performance. However, we will limit the statistical analysis to *MCC* as it is considered one of the most valuable and unbiased metrics to compare prediction models statistically [48].

C. $\boldsymbol{RQ}_3$ - *Complementarity between the PDG metrics-based model and the baseline model*

Upon addressing $\mathbf{RQ}_2$ and assessing the performance of the defect prediction model based on PDG metrics, we will compare it with the existing baseline developed by Dalla Palma et al. [14]. Specifically, we will run the baseline and conduct a complementarity analysis to understand the overlap with the PDG-based model. Given the two prediction models, m_i and m_j, we will compute (1) the number of bugs correctly predicted by both m_i and m_j, (2) the number of bugs correctly predicted by m_j and missed by m_i, (3) the number of bugs correctly predicted by m_i only and missed by m_j, and (4) the number of bugs missed by both m_i and m_j. Such an analysis could provide insights into the complementarity of the two approaches, other than assessing the actual value of our model compared to the baseline. The overlap metrics will indicate the extent to which a combination of the model built in $\mathbf{RQ}_2$ and the baseline [14] would have the potential to improve further the performance of IaC defect prediction, i.e., the likelihood that our working hypothesis may be accepted. We will finally address our hypothesis through the next research question.

D. $\boldsymbol{RQ}_4$ - *Comparing the performance to the existing baseline and creating a "Hybrid" defect prediction model*

Upon assessing the complementarity between the PDG metrics-based and state-of-the-art metrics-based models, we will build and assess the performance of a "hybrid" defect prediction model that combines the metrics from the two individual models. We will explore the relative prediction power of the metric sets, i.e., *Best-PDG* from $\mathbf{RQ}_2$, and *ICO*, *Delta*, and *Process* experimented by Dalla Palma et al. [14]. The four metrics sets will be combined to construct 15 different models: *"Best-PDG"*, i.e., the best model coming from $\mathbf{RQ}_2$, *"ICO"*, i.e., the best model coming from the work by Dalla Palma et al. [14], *"Delta"*, *"Process"*, *"Best-PDG + ICO"*, *"Best-PDG + Delta"*, *"Best-PDG + Process"*, *"Delta + Process"*, *"Delta + ICO"*, *"Process + ICO"*, *"Best-PDG + ICO + Delta"*, *"Best-PDG + ICO + Process"*, *"Best-PDG + Delta + Process"*, *"ICO + Delta + Process"*, *"Total"*. In doing so, we will compare various combinations of metric sets with respect to the individual models only relying on PDG, structural, and process metrics, respectively. We will compute RFE to score the different feature subsets and select the best-scoring collection of features. Finally, we will employ the same evaluation metrics as $\mathbf{RQ}_2$, i.e., *precision*, *recall*, *F-Measure*, *MCC*, and *AUC-PR*.

E. Addressing Our Working Hypothesis

To finally address our working hypothesis, we first define a formal null hypothesis, namely:

> ***Hn***$_1$. *There is no statistically significant difference between the MCC achieved by any of the models relying on PDG metrics and the those defined by Dalla Palma et al. [14].*

According to the null hypothesis, should any of the models featuring PDG metrics obtain performance not statistically better than the state-of-the-art model, then we would have failed to demonstrate that PDG metrics may improve IaC defect prediction. On the contrary, we would accept the alternative hypothesis, which is:

[2] Available at: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html

> ***An***$_1$. *The MCC achieved by any of the models relying on PDG metrics is statistically better than the those defined by Dalla Palma et al. [14].*

To address the hypothesis, we will compute the ρ-value from a matched pair Wilcoxon's rank test [46] for all pairs of techniques, applying a post-hoc Bonferroni's correction [29] to deal with the multiple comparisons. Should the ρ-value be lower than 0.05, we will reject the null hypothesis and accept the alternative ones, i.e., the PDG metrics improve the performance of IaC defect prediction models. Otherwise, we will accept the null hypothesis. In addition, we will also compute the Cohen's δ effect size measure [12] to assess the magnitude of the differences observed.

F. Publication of Generated Data

All the data generated from our study will be publicly available in an online repository on FIGSHARE. We also plan to release the scripts next to the data collected and used for the statistical analysis we will present in the paper.

V. LIMITATIONS

This section discusses the potential threats that may affect the validity of our empirical study plan [47].

Threats to construct validity. The first possible threat concerns the projects we will analyze in our study. We will rely on publicly available resources built in the context of previous research [14] that have already been used and validated, making us confident of the reliability of the selected projects. Another threat concerns how we will collect the set of PDG metrics. We plan to use the PDG builder that has already been used and validated [31], [32]. We will attempt to perform manual investigations on a statistically significant sample of PDG slices to assess the degree of accuracy of the extracted metrics (Section III-C2)—in this way, we will be able to provide indications of the confidence level of our conclusions.

Threats to internal validity. Threats to *internal* validity concern internal factors we might not consider that could affect the investigated variables. In particular, the choice of metrics might positively or negatively influence the classification. We will mitigate this threat by considering a comprehensive set of PDG metrics gathered from the literature [2], [34]. Similarly, data balancing is a critical aspect of defect prediction, so we plan to evaluate several over- and under-sampling techniques and how they could affect the model's performance.

Threats to external validity. Threats to *external* validity concern the generalization of results. First, we will analyze 135 Ansible-based systems from different application domains and with different characteristics (e.g., number of contributors, size, number of commits, etc.). Second, our proposal revolves around within-project defect prediction, so we will learn features that characterize failure-prone IaC scripts from the individual projects considered. Projects with a small number of defective instances will be discarded in this context: indeed, the absence of defects would not allow any machine learner to distinguish failure-prone from failure-free scripts. Finally, another threat is related to the classifier selection. We will evaluate five classifiers widely used in previous studies on bug prediction (e.g., [7], [14], [15]).

Threats to conclusion validity. Concerning the relationship between treatment and outcome, we will exploit a set of widely-used metrics to evaluate the performance of defect prediction techniques (i.e., precision, recall, F-measure, MCC, AUC-PR) [14], [40], [41]. In addition, we will use appropriate statistical tests, i.e., the Wilcoxon Test and the Cliff's Delta, which will allow us to support our findings and address our hypothesis. When assessing the contribution of the features to use in our approach, we will rely on the *Recursive Feature Elimination* algorithm [10], which the research community has used for the same purpose [1], [14]. Furthermore, since we will exploit change-history information to compute the PDG metrics, our study's evaluation design differs from the k-fold cross-validation generally exploited while evaluating defect prediction techniques. In particular, we will use the whole history of a system for the evaluation by adopting a walk-forward validation and assuring that new data (i.e., new releases) used to evaluate the model were never antecedent to those used to train it.

Another potential limitation concerns the intrinsic lifecycle of IaC defects: they must be reported and fixed before their introducing change is known. Our research will leverage the SZZ algorithm and commit messages indicating defect-fixing activities to mine defect data. As such, we acknowledge that undocumented defects, i.e., defects not reported in the issue tracker, could lead to classifying failure-prone scripts as "neutral" mistakenly.

VI. RELATED WORK

In the last few years, IaC has received increasing attention in the research community due to the paradigm shift in software design and development. Various works have investigated the literature, exploring the adoption, challenges, and defects of IaC [11], [21], [24], [35], [37]. Jiang and Adams [23] investigated the co-evolution between infrastructure and production code, finding that the infrastructure code is coupled with test files, leading testers to change infrastructure specifications often during continuous improvement. Analyzing the literature on smelliness in IaC, we found a plethora of studies. Sharma et al. [43] looked for code smells in the source code of configuration management tools, proposing a catalog of eleven design configuration smells. Opdebeeck et al. [31] proposed six code smells related to Ansible variable precedence rules and lazy-evaluated template expressions. Rahman et al. [38] proposed a catalog of seven security smells in IaC, extracted from Puppet script in open-source repositories. Later, Rahman et al. [39] proposed SLAC, a tool that checks for security smells in Ansible. Opdebeeck et al. [32] proposed GASEL, a security smell detector that recognizes seven security smells in Ansible scripts. Other studies related to Ansible concern the use of deep learning to detect anti-patterns and inconsistencies in the naming of tasks [6], and the application of change

distilling to study and predict version increments in open-source Ansible roles [33].

Moving on to defect prediction studies in IaC, Van der Bent et al. [44] defined a model to assess the quality of Puppet code. Dalla Palma et al. [14] proposed a framework to help practitioners to predict failure-prone in Ansible scripts focusing on several structural and process metrics [13] and reaching performance around 70%. Finally, Rahman et al. [40], [41] used text and source code properties to construct defect prediction models reaching performance around 70%.

Our registered report can be seen as complementary to the work mentioned above, as it aims at studying the impact of the program dependence graph characteristics when predicting the failure-proneness of IaC scripts. Moreover, we will focus on Ansible projects instead of Puppet.

VII. Conclusion

The ultimate goal of our research is to assess whether a IaC defect prediction model relying on PDG metrics can improve on a state-of-the-art defect predictor based on structural and process metrics. We will start working toward this goal by computing the PDG metrics on a set of 137 Ansible projects. Then we will employ a defect prediction model to address the goals of our investigation and, based on the conclusions we will be able to draw and finally provide actionable items and implications for researchers and practitioners.

References

[1] A. Adorada, P. W. Wirawan, K. Kurniawan, et al. The comparison of feature selection methods in software defect prediction. In *2020 4th International Conference on Informatics and Computational Sciences (ICICoS)*, pages 1–6. IEEE, 2020.

[2] B. S. Alqadi and J. I. Maletic. Slice-based cognitive complexity metrics for defect prediction. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 411–422, 2020.

[3] G. Ayyappan and S. Karpagam. Analysis of a bulk service queue with unreliable server, multiple vacation, overloading and stand-by server. *International Journal of Mathematics in Operational Research*, 16(3):291–315, 2020.

[4] G. E. Batista, R. C. Prati, and M. C. Monard. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD explorations newsletter*, 6(1):20–29, 2004.

[5] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[6] N. Borovits, I. Kumara, D. Di Nucci, P. Krishnan, S. D. Palma, F. Palomba, D. A. Tamburri, and W.-J. v. d. Heuvel. Findici: Using machine learning to detect linguinformation and software technologyic inconsinformation and software technologyencies between code and natural language descriptions in infrastructure-as-code. *Empirical Software Engineering*, 27(7):178, 2022.

[7] D. Bowes, T. Hall, and J. Petrić. Software defect prediction: do different classifiers find the same defects? *Software Quality Journal*, 26:525–552, 2018.

[8] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[9] S. Chaisiri, R. Kaewpuang, B.-S. Lee, and D. Niyato. Cost minimization for provisioning virtual servers in amazon elastic compute cloud. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 85–95. IEEE, 2011.

[10] X.-w. Chen and J. C. Jeong. Enhanced recursive feature elimination. In *Sixth International Conference on Machine Learning and Applications (ICMLA 2007)*, pages 429–435. IEEE, 2007.

[11] M. Chiari, M. De Pascalis, and M. Pradella. Static analysis of infrastructure as code: a survey. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 218–225. IEEE, 2022.

[12] J. Cohen. The effect size index: d. statinformation and software technologyical power analysis for the behavioral sciences. *Abingdon-on-Thames: Routledge Academic*, 1988.

[13] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri. Toward a catalog of software quality metrics for infrastructure code. *Journal of Systems and Software*, 170:110726, 2020.

[14] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri. Within-project defect prediction of infrastructure-as-code using product and process metrics. *IEEE Transactions on Software Engineering*, 48(6):2086–2104, 2021.

[15] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, 2017.

[16] R. O. Duda, P. E. Hart, et al. *Pattern classification and scene analysis*. A Wiley-Interscience publication. Wiley, 1973.

[17] C. Ebert, S. Technologyof, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[18] J. Ewart, M. Marschall, and E. Waud. *Chef: Powerful Infrastructure Automation*. Packt Publishing Ltd, 2017.

[19] D. Falessi, J. Huang, L. Narayana, J. F. Thai, and B. Turhan. On the need of preserving order of data when validating within-project defect classifiers. *Empirical Software Engineering*, 25:4805–4830, 2020.

[20] Y. Freund and L. Mason. The alternating decision tree learning algorithm. In *icml*, volume 99, pages 124–133. Citeseer, 1999.

[21] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 580–589. IEEE, 2019.

[22] L. Hochstein and R. Moser. *Ansible: Up and Running: Automating configuration management and deployment the easy way*. " O'Reilly Media, Inc.", 2017.

[23] Y. Jiang and B. Adams. Co-evolution of infrastructure and source code-an empirical study. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 45–55. IEEE, 2015.

[24] I. Kumara, M. Garriga, A. U. Romeu, D. Di Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel. The do's and don'ts of infrastructure code: A systematic gray literature review. *Information and Software Technology*, 137:106593, 2021.

[25] Y. Kurniawan. *Ansible for AWS*. Leanpub, 2016.

[26] M. P. LaValley. Logistic regression. *Circulation*, 117(18):2395–2399, 2008.

[27] J. Loope. *Managing infrastructure with puppet: configuration management at scale*. " O'Reilly Media, Inc.", 2011.

[28] K. Morris. *Infrastructure as code*. O'Reilly Media, 2020.

[29] M. A. Napierala. What is the bonferroni correction? *Aaos Now*, pages 40–41, 2012.

[30] W. S. Noble. What is a support vector machine? *Nature biotechnology*, 24(12):1565–1567, 2006.

[31] R. Opdebeeck, A. Zerouali, and C. De Roover. Smelly variables in Ansible infrastructure code: detection, prevalence, and lifetime. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 61–72, 2022.

[32] R. Opdebeeck, A. Zerouali, and C. De Roover. Control and data flow in security smell detection for infrastructure as code: Is it worth the effort? In *Proceedings of the 20th International Conference on Mining Software Repositories (MSR 2023)*, pages 534–545, 2023.

[33] R. Opdebeeck, A. Zerouali, C. Velázquez-Rodríguez, and C. De Roover. On the practice of semantic versioning for ansible galaxy roles: An empirical study and a change classification model. *Journal of Systems and Software*, 182:111059, 2021.

[34] K. Pan, S. Kim, and E. J. Whitehead, Jr. Bug classification using program slicing metrics. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 31–42, 2006.

[35] A. Rahman, E. Farhana, C. Parnin, and L. Williams. Gang of eight: A defect taxonomy for infrastructure as code scripts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 752–764, 2020.

[36] A. Rahman, E. Farhana, and L. Williams. The 'as code' activities: Development anti-patterns for infrastructure as code. *Empirical Software Engineering*, 25:3430–3467, 2020.

[37] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams. A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108:65–77, 2019.

[38] A. Rahman, C. Parnin, and L. Williams. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175. IEEE, 2019.

[39] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams. Security smells in ansible and chef scripts: A replication study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(1):1–31, 2021.

[40] A. Rahman and L. Williams. Characterizing defective configuration scripts used for continuous deployment. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 34–45. IEEE, 2018.

[41] A. Rahman and L. Williams. Source code properties of defective infrastructure as code scripts. *Information and Software Technology*, 112:148–163, 2019.

[42] J. Schwarz, A. Steffens, and H. Lichter. Code smells in infrastructure as code. In *2018 11Th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 220–228. IEEE, 2018.

[43] T. Sharma, M. Fragkoulis, and D. Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 189–200, 2016.

[44] E. Van der Bent, J. Hage, J. Visser, and G. Gousios. 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)*, pages 164–174. IEEE, 2018.

[45] R. S. Wahono. A systematic literature review of software defect prediction. *Journal of software engineering*, 1(1):1–16, 2015.

[46] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

[47] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[48] J. Yao and M. Shepperd. The impact of using biased performance metrics on software defect prediction research. *Information and Software Technology*, 139:106664, 2021.