UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

TESI DI LAUREA

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

RELATORE

Prof. Dario Di Nucci

Dott.ssa Valeria Pontillo

Università degli Studi di Salerno

CANDIDATO

Gerardo Iuliano

Matricola: 0522501329

Anno Accademico 2022-2023

Questa tesi è stata realizzata nel

Dedicated to all those who have embraced the world of computing with curiosity and passion, relentlessly seeking innovative solutions. May this thesis be a modest contribution to the vast landscape of computer science knowledge, inspiring new challenges and discoveries.

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 extracted 11 PDG metrics from 139 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 assessed the performance of a combined model that mixes together PDG and existing IaC-oriented metrics.
Results: The most occurring predictors are MAXPDGVERTICES, EDGESTOVERTICESRATIO, EDGESCOUNT, and VERTICESCOUNT. Program Dependence Graph metrics-based models trained using RANDOM FOREST and DECISION TREE perform statistically better than those relying on the remaining classifiers. PDG metrics-based models correctly predicted the number of bugs over 20% more than Delta and Process metrics-based models. Finally, PDG metrics can improve the performance of Delta and Process metrics. However, such metrics have negligible effects on models employing ICO metrics.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Application Context

DevOps [1] 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 [2]. Configuration management tools, e.g., Ansible [3], Chef [4], and Puppet [5], 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 [6], with Ansible emerging as one of the most popular solutions [7]. Nonetheless, Infrastructure as Code remains source code that may exhibit the same flaws as application code, e.g., code smells [8, 9, 10, 11], defects [12, 13, 14, 15], or security concerns [16]. These issues may notably impact the reliability and maintainability of infrastructure code and have several negative implications, e.g., hot service stand-by [17] or costly elastic provisioning [18].

More specifically, defect prediction [19] 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 embrace 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. [20], 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.

1.2 Motivations and Objectives

While the empirical study conducted on RADON showed that it may predict defect-prone scripts with high accuracy, Dalla Palma et al. [20] also pointed out that further metrics able to characterize orthogonal properties of Ansible scripts may further improve defect prediction capabilities. A Program Dependence Graph (PDG) representation captures both the control and data flow of Ansible scripts, providing an improved mechanism to analyze the inner workings of those scripts.

This thesis overviews our research method to address our hypothesis: we propose a *confirmatory empirical investigation* where we (1) collected 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) devised a new defect prediction model based on the newly defined metrics; and (3) evaluated 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. [20]. We finally assessed statistically whether PDG metrics contribute to the prediction of defective IaC scripts.

1.3 Results Obtained

We conducted several experiments, achieving the following results. The PDG metrics that maximize the performance of defect prediction models are *maxPDGVertices*, *verticesCount*, *edgesToVerticesRatio* and *edgesCount* with a mean rank value of 2.21, 2.92,

3.65 and 3.88 respectively. The Program Dependence Graph metrics-based models trained using *Random Forest* and *Decision Tree* perform statistically better than those relying on the remaining classifiers. Program Dependence Graph metrics improve the number of bugs correctly predicted by 23.8% over Delta metrics alone, 21.13% over Process metrics alone, and 1.6% over ICO metrics alone. Finally, PDG metrics can improve the performance of Delta and Process metrics. However, such metrics have negligible effects on models employing ICO metrics.

1.4 Structure of Thesis

The structure of the thesis is organized into seven chapters, each serving a specific purpose and contributing to the overall understanding of the research conducted. Here's a brief explanation of each chapter:

This chapter provides an overview of the context in which the research is conducted. It outlines the reasons and goals behind the research and discusses any preliminary results or findings.

Chapter 2 discusses the background and concepts related to *DevOps* and *Infrastructure as Code*. It also provides information about machine learning and its application in defect prediction and introduces the concept of *Program Dependence Graph* as a key element in the research.

Chapter 3 suggests the application of concepts and tools related to program dependency graphs to evaluate and improve Ansible configurations in terms of quality and performance.

Chapter 4 lists the research questions addressed in the study. In addition, it explains how the research context was selected, details the variables used in the empirical study, and discusses the use of machine learning in defect prediction. Finally, it outlines the research questions to be tested.

Chapter 5 presents the results and analysis for each of the research questions and discusses how the results relate to the working hypothesis.

Chapter 6 combines critical analysis of the research findings, a recognition of the study's limitations, and a discussion of the broader implications and significance of the research in its respective field.

Chapter 7 addresses various threats to the validity of the research, including construct validity, internal validity, external validity, and conclusion validity.

Finally, Chapter 8 summarizes the main findings and concludes the thesis.

CHAPTER 2

Background

2.1 DevOps and Infrastructure-as-Code

The DevOps methodology is drastically changing the way software is designed and managed. DevOps involves in adoption of a set of organizational and technical practices, e.g., continuous integration, continuous deployment, blending development, and operation teams, in order to thrive and remain competitive in the modern digital ecosystem and market, which demands fast and early releases, continuous software updates, constant evolution of market needs, and adoption of scalable technologies such as Cloud computing. Infrastructure-as-Code (IaC) is a DevOps practice that involves using machine-readable code to describe complex, often Cloud-based deployments. Cloud computing has been the primary driving force behind Infrastructure-as-Code (IaC), as it has made the programmatic provisioning, configuration, and management of computational resources a widely adopted practice. Many languages and platforms have been developed, each dealing with specific aspects of infrastructure management, from tools able to provision and orchestrate virtual machines (Terraform, Cloudify), to those doing a similar job for container technologies (Kubernetes, Docker Swarm), to machine image management tools (Packer), to configuration management tools (Chef, Puppet, Ansible). In recent years, Ansible has

gained popularity as a simple and agent-less (i.e., no master node) alternative to other more complex IaC technologies such as Chef and Puppet. Ansible is an automation engine that uses the YAML language to automate a variety of tasks, including cloud provisioning, configuration management, and application deployment. It operates by connecting to nodes and deploying *Ansible modules*, which are scripts that describe or modify the state of the system. Ansible executes these modules on an as-needed basis. While Ansible modules ensure the proper functionality of Ansible scripts, *playbooks* are critical for orchestrating multiple components of the infrastructure topology with precise control over scalability (i.e., how many machines to handle at a time). Playbooks are essential for configuration management and multi-machine deployment in Ansible, as they can declare configurations and coordinate the steps of any manual, ordered process by executing tasks within one or more *plays*. Each play maps hosts to specific roles, represented by Ansible *tasks* that collectively invoke Ansible modules.

```
---
-   name: Update web servers
    hosts: webservers
    remote_user: root

    tasks:
        -   name: Ensure apache is at the latest version
            ansible.builtin.yum;
                name: httpd
                state: latest
        -   name: Write the apache config file
            ansible.builtin.template:
                src: /srv/httpd.j2
                dest: /etc/httpd.conf

-   name: Update db servers
    hosts: databases
    remote_user: root

    tasks:
        -   name: Ensure postgresql is at the latest version
            ansible.builtin.yum;
```

```
23                name: postgresql
24                state: latest
25        -   name: Ensure that postgresql is started
26            ansible.builtin.service:
27                name: postgresql
28                state: started
```

Listing 2.1: An example of Ansible code.

For example, Listing 2.1 shows an Ansible code snippet representing a playbook that provisions and deploys a website. To this aim, it configures various aspects such as the ports to open on the host container, the name of the user account, and the desired database to deploy. It first targets the web servers to ensure that the Apache server is at the latest version, and then the database servers to ensure that PostgreSQL is at the latest version and started. It achieves this by mapping the hosts to their respective tasks. There, `yum` and `service` are modules to manage packages with the yum package manager and to control services on remote hosts, respectively; `name` (i.e., the name of the package and the database) and `state` (i.e., whether present, absent or otherwise) are parameters of these modules. In short, by composing a playbook of multiple plays, it is possible to orchestrate multi-machine deployments and run specific commands on them in the `webservers` and `databases` groups.

2.2 Machine Learning and Defect Prediction

Machine Learning is a subfield of artificial intelligence (AI) that focuses on the development of algorithms and models that enable computer systems to learn and make predictions or decisions without being explicitly programmed. It involves the use of statistical techniques to enable computers to learn from and analyze large amounts of data.

In machine learning, algorithms are trained on data to recognize patterns, make predictions, or take actions based on that data. The learning process involves adjusting the parameters or internal representations of the algorithms to improve their performance over time. This is typically done through the use of labeled training data, where the algorithm is provided with input data and the corresponding correct

output or label. The algorithm then learns to map the input data to the correct output by identifying patterns and relationships within the data.

Machine learning can be categorized into several types, including supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning. Supervised learning involves training algorithms using labeled data, while unsupervised learning involves finding patterns and structures in unlabeled data. Semi-supervised learning combines elements of both supervised and unsupervised learning, while reinforcement learning involves training algorithms through interaction with an environment to maximize rewards.

Machine learning has a wide range of applications, including image and speech recognition, natural language processing, recommendation systems, fraud detection, autonomous vehicles, medical diagnosis, and many others. It has become an essential tool in various industries and continues to advance with the availability of large datasets, improved algorithms, and increased computational power.

Supervised Learning is a type of machine learning where an algorithm learns from labeled training data to make predictions or decisions. In supervised learning, the training data consists of input samples and their corresponding desired outputs or labels. The goal of supervised learning is to train a model or algorithm to generalize and accurately predict the correct output for new, unseen input data. During the training phase, the algorithm learns the relationship between the input features and the corresponding labels by minimizing the difference between its predicted outputs and the true labels in the training data. Supervised learning can be further divided into two main categories:

Regression. In regression tasks, the goal is to predict a continuous numerical value as the output. The algorithm learns a mapping between the input features and a continuous target variable. For example, predicting the price of a house based on its features such as size, number of rooms, and location.

Classification. In classification tasks, the goal is to predict a discrete class label as the output. The algorithm learns to classify input data into predefined categories or classes. For example, classifying emails as spam or not spam based on their content and attributes.

Defect Prediction refers to the process of using historical data and machine learning techniques to estimate or predict the likelihood of defects or bugs in software modules or projects. It aims to identify areas of code that are more likely to contain defects, allowing developers to allocate resources more effectively for testing and maintenance activities. Defect prediction can help software development in several ways, including:

Early identification of high-risk areas. Developers can focus their testing and debugging efforts on code modules predicted to have a higher likelihood of defects, thereby improving software quality.

Resource allocation. By predicting defect-prone areas, development teams can allocate resources effectively, targeting code segments that require more attention, testing, or code review.

Maintenance and software evolution. Defect prediction models can aid in identifying areas of code that are more likely to introduce defects during software updates, helping developers plan maintenance activities accordingly.

2.3 Program Dependence Graph

A Program Dependence Graph is a graphical representation that depicts the relationships and dependencies between components and entities within a program. It provides a visual representation of how the program's statements, functions, or modules interact with each other. It captures two types of dependencies:

Control Flow Dependencies. Control flow dependencies represent the order in which program statements are executed. They illustrate how the program flows from one statement to another based on conditions, loops, branches, or function calls. Control flow dependencies indicate the sequence of execution and the decision points within the program.

Data Flow Dependencies. Data flow dependencies represent the flow of data or information between different parts of the program. They show how variables

or data values are produced, consumed, or transformed within the program. Data flow dependencies can be used to track how variables are read, written, or passed as parameters between functions or modules.

In a program dependency graph, nodes represent program components such as statements, functions, or modules, and edges represent the dependencies between them. Control flow dependencies are typically represented by directed edges, indicating the flow of execution, while data flow dependencies can be represented by labeled edges indicating the flow of data values or variables.

Tools and techniques, such as static code analysis or program slicing, can be used to construct program dependency graphs automatically. These graphs can be generated at different granularities, ranging from high-level representations of the entire program to more detailed representations of specific functions or modules.

Overall, program dependency graphs serve as a powerful visualization tool for understanding and analyzing the structure, behavior, and dependencies of a program, enabling developers to gain insights and make informed decisions during program comprehension, maintenance, or optimization tasks.

CHAPTER 3

Program-Dependence-Graph-based Metrics for Ansible

This chapter describes how the process of extracting PDGs at the task level was performed starting from PDGs at the repository level. Metrics are also analyzed and described.

3.1 Task-level Program Dependence Graphs for Ansible

The PDG was extracted using the PDGs proposed by Opdebeeck et al. [8, 21], who proposed a Program Dependence Graph (PDG) representation for Ansible scripts. Their PDG is a graph that captures the control flow and data flow of an Ansible script. The nodes of the graph represent elements of the script, such as tasks, expressions, variables, and data literals. The edges represent control flow (between tasks) or various types of data flow (between data and tasks).

The tool extracts the whole-project PDG, this means that the resulting graph represents the entire structure of the repository, such as modules, roles, playbooks, and inventory. To extract metrics based on the PDG, we need to work at a lower level. 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 [22, 23], which considers 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. Thus, given the nature of the metrics we intend to extract, we need to work at the task level, the equivalent of slices in procedural code.

To work at a task level, we need to extract task-level PDGs from the repository-level PDG extracted by the tool. To extract PDG at the task level, we implemented an algorithm. The algorithm is an exploration algorithm that starts from the task node and collects all the information about the task, e.g., the activating condition, the variables used, the variables changed, the task that precedes it, the task that succeeds it, the Ansible modules used, the parameters, and more.

Note that we cannot build a PDG for single files as this may incorrectly approximate data flow, as pointed out by Opdebeeck et al. [8].

An algorithm to extract PDG for Ansible. Below is a description of the algorithm proposed by Opdebeeck. The algorithm takes as input the repository-level PDG and the target task node against which we want to extract a task-level PDG, i.e., a subgraph of the source graph.
The algorithm uses data flow edges to determine which nodes should be included in the subgraph. While it uses control flow edges to determine the predecessors and successors of the task, these edges are then used only to determine the control flow of the target task.
At distance 1 from the target task node, most of the nodes of interest are present. In particular, via control flow edges, can be reached the nodes of the predecessor and successor task and the `condition` node, which contains the task activation condition. The `condition` node, in turn, is connected with other nodes such as the `variable` or `expression` nodes, that contribute to the value of the condition. Given the importance of having the variables that contribute to task activation available, we decided to implement different behavior for nodes with distance 1 than for nodes with greater distance.
At distance 1, both incoming and outgoing edges are considered. Also at this distance,

if a `condition` node reached by a control flow edge is present, it is added into the subgraph and is used to continue the exploration since it is a node that is of interest to our task and contains relevant information for calculating the metrics. All other nodes reached by a control flow edge that are not condition nodes are included in the subgraph to keep track of the control flow but are not used to continue exploration. The rest of the nodes reached instead by data flow edges are inserted into the subgraph and are, in turn, used to continue exploration.

At distances greater than 1, instead, only incoming data flow edges are considered. This is because it is necessary to trace the entire data flow that is used in the task.

The return value of the algorithm is a list of nodes. Using the NETWORKX library and the *subgraph* method we can now extract from the source graph a subgraph consisting exclusively of the nodes marked by the algorithm.

```
function extractPDGTaskLevel(G, S):
    V = G.nodes
    E = G.edges(data=True)
    source, target, keys = unzip(E)
    workList, markList = set()
    workList = S
    w = workList.pop()
    markList.add(w)
    #distance 1 from task node
    for i in range(0, length(V)):
        if target[i] equals w:
            if source[i] is not in markList:
                if keys[i] is "CONTROL_FLOW_EDGE":
                    if node_type of source[i] is "Conditional":
                        add source[i] to workList
                    else:
                        add source[i] to markList
                else:
                    add source[i] to workList
        if source[i] equals w:
            if target[i] is not in markList:
                if keys[i] is "CONTROL_FLOW_EDGE":
                    add target[i] to markList
                else:
```

```
                    add target[i] to workList
    while length(workList) is not 0:
        w = workList.pop()
        add w to markList
        for i in range(0, length(V)):
            if target[i] equals w:
                if source[i] is not in markList:
                    if keys[i] is "CONTROL_FLOW_EDGE":
                        add source[i] to markList
                    else:
                        add source[i] to workList
    return markList
```

Listing 3.1: Pseudocode of the algorithm that extracts a task-level PDG starting from a repository-level PDG

The following Listing 3.2 is an ansible task extracted from a playbook of a repository in our study. While Figure 3.1 is the representation of the task using the Program Dependence Graph.

```
-   name: Gather Distribution Info
    ansible.builtin.setup:
        gather_subset: distribution!
    when:
        - ansible_distribution is not defined
```

Listing 3.2: Ansible task.

Figure 3.1 is a graphical representation of the Program Dependence Graph of Listing 3.2. Due to the large amount of information contained in the PDG, not all information has been included in the graphical representation. The structure of the PDG is contained in an XML document in the GraphML format. GraphML is a standard format for representing the data of a graph.

```
<graph edgedefault="directed">
    <data key="d0">RHEL7-STIG</data>
    <data key="d1">latest</data>
    <node id="746">
      <data key="d2">Task</data>
```

```
      <data key="d5">"ansible.builtin.setup"</data>
      <data key="d6">"Gather distribution info"</data>
    </node>
    <node id="747">
      <data key="d2">Variable</data>
      <data key="d6">"ansible_distribution"</data>
    </node>
    <node id="748">
      <data key="d2">Expression</data>
      <data key="d10">"ansible_distribution is not defined"</data>
    </node>
    <node id="749">
      <data key="d2">IntermediateValue</data>
    </node>
    <node id="750">
      <data key="d2">Conditional</data>
    </node>
    <node id="751">
      <data key="d2">Literal</data>
      <data key="d13">"str"</data>
      <data key="d14">"distribution"</data>
    </node>
    <node id="752">
      <data key="d2">Task</data>
      <data key="d5">"ansible.builtin.assert"</data>
      <data key="d6">"Check OS version and family"</data>
    </node>
    <edge source="746" target="752">
      <data key="d15">ORDER</data>
    </edge>
    <edge source="747" target="748">
      <data key="d15">USE</data>
    </edge>
    <edge source="748" target="749">
```

```
      <data key="d15">DEF</data>
    </edge>
    <edge source="749" target="750">
      <data key="d15">USE</data>
    </edge>
    <edge source="750" target="746">
      <data key="d15">ORDER</data>
    </edge>
    <edge source="750" target="752">
      <data key="d15">ORDER</data>
    </edge>
    <edge source="751" target="746">
      <data key="d15">KEYWORD</data>
      <data key="d19">"args.gather_subset"</data>
    </edge>
</graph>
```

Listing 3.3: Essential information of Ansible task in GraphML format.

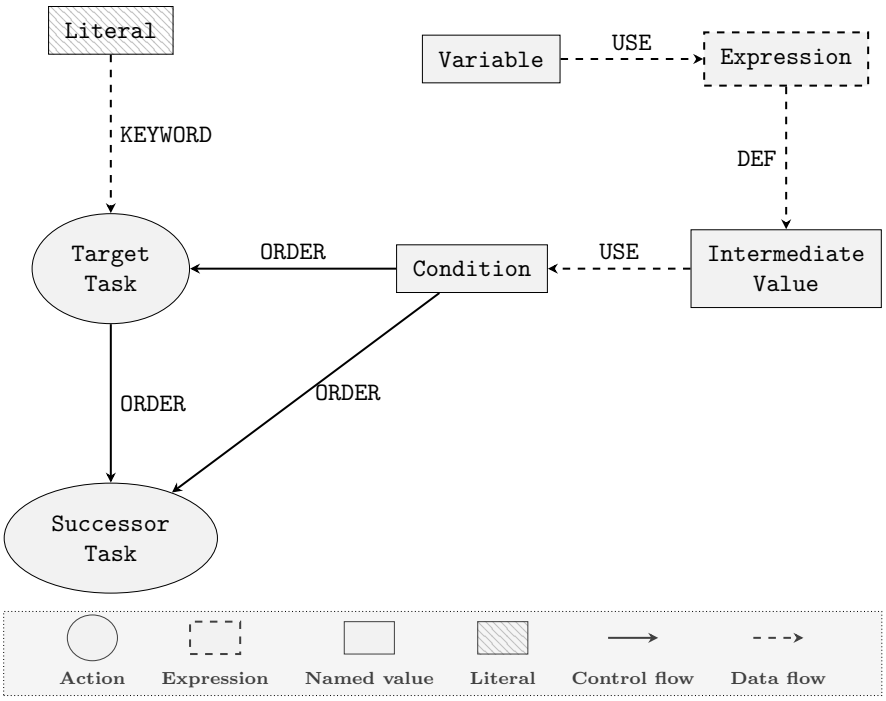

Figure 3.1: Program dependence graph for the example of Listing 3.2.

The task-level pdg extraction algorithm was integrated into a specific tool. It

takes as input a repository and its related PDG and returns the repository with all its task-level PDGs. The output repository preserves the directory structure, while each file containing one or more tasks is decomposed into many files, one per task. These files are simply the representation of the task by PDGs in GraphML format.

To validate the implemented PDG slicer extractor, we performed a manual analysis. We selected a statistically significant sample of the PDG slices (confidence level=95%, margin of error=5%) and, for each slice, we manually compared the slice to the file from which it originates to validate the slicer.
Below we are going to show how metrics were extracted from each task-level pdg and how we moved from task-level metrics to file-level metrics.

3.2 PDG-based Metrics for Ansible

The metrics we are going to extract are listed in the Table 3.1. However, during the implementation, some inconsistencies emerged. In particular, as we mentioned in the design, the metrics *taskCount*, *taskCoverage*, *taskSize*, *taskSpatial* and *taskIdentifier* were discarded because they represent concepts close to some ICO metrics. in addition, *pdgVerticesSum* have a similar definition to *verticesCount*, and was therefore discarded. From the 17 metrics collected, after careful analysis, we reduced the metrics to 11.

The metrics for which an extractor has been implemented are: *maxPdgVertices*, *lackOfCohesion*, *verticesCount*, *edgesCount*, *edgesToVerticesRatio*, *globalInput*, *globalOutput*, *directFanIn*, *indirectFanIn*, *directFanOut*, *indirectFanOut*

Before proceeding with the extraction of task-level metrics, it was necessary to develop a docker image that would allow the tool proposed by Opdebeeck et al. [8, 21] to work together with the PDG slicer extractor. The docker image consists of a series of steps:

1. Checkout to the first commit in the time order that the target repository has received during its lifecicle.

2. Launch the tool to extract the PDG of the entire project.

3. Launch the tool to extract task-level PDG.

4. Calculate metrics for each PDG at a task level.

5. Checkout to the next commit in the time order that the target repository has received. Restart from step 2 until the last commit.

Table 3.1: Metrics leveraging Program Dependence Graphs in functional programming contextualized to Infrastructure as Code

Orig. PDG Metric	Description	IaC-PDG Metric	Description
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.
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.
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.

CHAPTER 4

Research Methodology

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.

4.1 Research Questions

Our empirical investigation aims to answer 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 $\mathbf{RQ}_1$, we seek to understand which metrics related to the program dependence graph contribute the most to detecting defects in IaC scripts. These observations were 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 $\mathbf{RQ}_2$, we employed the most promising metrics coming from $\mathbf{RQ}_1$ in experimentation aimed at establishing the best machine learning model relying on PDG metrics. Then we performed a further step ahead, namely that of understanding how complementary the model coming from $\mathbf{RQ}_2$ with respect to the state-of-the-art model proposed by Dalla Palma et al. [20]. The outcome of $\mathbf{RQ}_3$ revealed insights into the potential added value of the model based on PDG metrics: such potential was finally quantified in $\mathbf{RQ}_4$, where we experimented with how different feature sets behave independently from each other and how they augment each other. As a last step, we addressed our working hypothesis by defining a more specific pair of null and alternative hypotheses, whose validity was statistically addressed. To design and report our study, we followed the empirical software engineering guidelines by Wohlin et al. [24], other than the *ACM/SIGSOFT Empirical Standards*.[1]

4.2 Context Selection

We collected data from the dataset of 139 open-source Ansible projects, publicly available on GITHUB and released by Dalla Palma et al. [20]. The dataset allowed us to compare the performance of models trained using structural, process, and PDG metrics. GitHub repositories satisfied specific criteria described in Table 4.1 and Metrics are grouped into four categories described in Table 4.2

The first step consisted of cloning all 139 repositories. During the cloning step, two repositories reported fatal errors that prevented the repositories from being cloned correctly. In particular, the Git tree of the repository appears to be corrupted, which prevents not only cloning but also the checkout operation. The repositories that have been successfully cloned are 137.

[1]Empirical Standards: `https://github.com/acmsigsoft/EmpiricalStandards`.

Table 4.1: GitHub repositories criteria

Criteria
The repository has at least one push event to its master branch in the last six months
The repository has at least two releases
At least 11% of the files in the repository are IaC scripts
The repository has at least two core contributors
The repository has evidence of continuous integration practice, such as the presence of a .travis.yaml file
The repository has a comments ratio of at least 0.2%
The repository has a commit frequency of at least 2 per month on average
The repository has an issue frequency of at least 0.023 events per month on average
The repository has evidence of a license, such as the presence of a LICENSE.md file
The repository has at least 190 source lines of code

Table 4.2: Metrics description

Metric	Description
IaC-Oriented	Metrics of structural properties derived from the source code of infrastructure scripts
Delta	Metrics that capture the amount of change in a file between two successive releases, collected for each IaC-oriented metric
Process	Metrics that capture aspects of the development process rather than aspects of the code itself. A description of the process metrics in this dataset can be found here
PDG Metrics	Metrics that capture aspects of the data flow and the control flow of infrastructure scripts.

4.3 Empirical Study Variables

The second 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.

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 rely on a binary value indicating the presence/absence of a bug.

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 [22, 23], 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 3.1 shows the complete set of metrics we will experiment with. The metrics was extracted using the PDGs proposed by Opdebeeck et al. [8, 21]. However, during the implementation, some inconsistencies emerged. In particular, the metrics *taskCount*, *taskCoverage*, *taskSize*, *taskSpatial* and *taskIdentifier* were discarded because they represent concepts close to some ICO metrics. In addition, *pdgVerticesSum* have a similar definition to *verticesCount*, and was therefore discarded. From the 17 metrics collected, after careful analysis, we reduced the metrics to 11.
The metrics for which an extractor has been implemented are: *maxPdgVertices*, *lackOfCohesion*, *verticesCount*, *edgesCount*, *edgesToVerticesRatio*, *globalInput*, *globalOutput*, *directFanIn*, *indirectFanIn*, *directFanOut*, *indirectFanOut*.

4.4 Machine Learning for Defect Prediction

The following shows how we leveraged machine learning classification.

Selecting Machine Learning Algorithms. In the context of our study, we experimented with multiple machine learning classifiers. First, we included *Naive Bayes* [25] and *Logistic Regression* [26] as classifiers that do not require much training data. Then, we considered *Decision Tree* [27], *Random Forest* [28], and *Support Vector Machine* [29], 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. [20] 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 [20]. 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. [20] 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.

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 [30]. This observation is particularly true in defect prediction, where the neutral class outnumbers the failure-prone class. We experimented with several under- and over-sampling configurations to overcome this obstacle. Specifically, we considered using the NEARMISS 1, NEARMISS 2, and NEARMISS 3 algorithms for the under-sampling. Finally, we experimented with a RANDOM UNDERSAMPLING approach that randomly explores the distribution of majority instances and under-samples them. As for the over-sampling, we experimented 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. [20]: as explained earlier, this choice allows us to compare our results with previous ones more fairly.

The training data was normalized, scaling numeric attributes. We plan to evaluate

Figure 4.1: Walk-forward validation process.

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 configured the hyper-parameters of the selected machine learning classifiers by using the RANDOM SEARCH algorithm [31], 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 developed the entire pipeline with the SCIKIT-LEARN library in PYTHON.

Validation of the Approach. To assess the performance of our models, we performed 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 was guided by a randomized search of the models' parameters through a walk-forward validation [32]. 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 was used as the training set, while the part to predict was used as the test set, preventing the test set has data antecedent to the training set. Afterward, the model performance was computed as the average of various runs. Figure 4.1 shows the validation process. Specifically, the number of iterations was equal to the number of parts minus one. We trained each model on the first n releases and tested on the (n+1)-th release.

4.5 RQ_1 - In Search of Suitable Program Dependency Graph Metrics for Defect Prediction Models

To evaluate the relative predictive power of the metrics listed in Table 3.1, we performed recursive feature selection to find the metrics that maximize the performance and rank them according to their importance for the prediction. Given an external estimator that assigns weights to features, recursive feature elimination (RFE) selected 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 was 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 applied cross-validation to score the different feature subsets and select the best-scoring collection of features. To this end, we employed the RFECV method[2] available in SCIKIT-LEARN.

4.6 RQ_2 - In Search of the Best Defect Prediction Model based on Program Dependency Graph Metrics

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

To address $\mathbf{RQ_2}$, we first computed 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 also computed 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. After-

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

ward, we established the best model by comparing the MCC of the experimented models through the Wilcoxon's rank test [33], applying the post-hoc Bonferroni's correction [34] to deal with the multiple comparisons that were performed during the validation process. In addition, we've also computed the Cohen's δ effect size measure [35] to assess the magnitude of the differences observed. We've still computed and discussed 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 reported statistics (i.e., mean, median, minimum, maximum, and standard deviation) about the classifier achieving the best performance. However, we've limited the statistical analysis to *MCC* as it is considered one of the most valuable and unbiased metrics to compare prediction models statistically [36].

4.7 $\mathbf{RQ}_3$ - Complementarity between the PDG Metrics-based Model and the Baselines

Upon addressing $\mathbf{RQ}_2$ and assessing the performance of the defect prediction model based on PDG metrics, we compared it with the existing baseline developed by Dalla Palma et al. [20]. Specifically, we ran the baseline and conducted a complementarity analysis to understand the overlap with the PDG-based model. Given the two prediction models, m_i and m_j, we computed (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 indicated the extent to which a combination of the model built in $\mathbf{RQ}_2$ and the baseline [20] 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 finally address our hypothesis through the next research question.

4.8 $\mathbf{RQ_4}$ - On the Performance of the Baselines and a Novel "Hybrid" Model

Upon assessing the complementarity between the PDG metrics-based and state-of-the-art metrics-based models, we built and assessed the performance of a "hybrid" defect prediction model that combines the metrics from the two individual models. We explored 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. [20]. The four metrics sets were 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. [20], *"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 compared various combinations of metric sets with respect to the individual models only relying on PDG, structural, and process metrics, respectively. We computed RFE to score the different feature subsets and select the best-scoring collection of features. Finally, we employed the same evaluation metrics as $\mathbf{RQ_2}$, i.e., *precision*, *recall*, *F-Measure*, *MCC*, and *AUC-PR*.

CHAPTER 5

Data analysis and results

5.1 Metrics Extraction

At the end of the docker image execution, we collected the results of the extraction; 137 repositories and all their respective commits were analyzed. The entire Dalla Palma [20] study dataset has 227,273 files, for each of which ICO metrics, Delta metrics, and Process metrics were extracted. Our extraction process was successful on 80 repositories. These 80 repositories represented 21,197 files out of the total 227,273. Of these 21,197 files, only 13,219 files were found to be suitable. The remaining files are files that do not contain tasks, configuration files, templates, and inventories. Therefore, since they had no relevant content for metrics calculation, they were ignored. The remaining 57 repositories for which metrics extraction was unsuccessful represent 206,075 files out of the total 227,273. The reason for this imbalance is the size of the repositories. Among the unanalyzed repositories, there are 6 repositories that are very large compared to the average. They account for about 140,000 files out of 227,273, thus more than half.

We focused on them to understand why they are not being analyzed:

- one repository had a problem on the working tree, thus not allowing checkouts.

- four repositories (redhat-openstack/infrared of 50,000 files, PGBlitz/PGBlitz.com of 47,000 files, valet-sh/valet-sh of 13,000 files and ceph/ceph-ansible of 11,000 files) were revealed to be poorly structured repositories that did not allow the tool to extract the PDG.
  ValueError: Could not auto-detect whether project at path is a role or a playbook

- one repository (openstax/cnx-deploy of 21,000 files) was processed by the PDG-tool but the result was an empty PDG.

We extracted metrics for 13,219 files out of 227,273.

5.2 RQ_1 - In Search of Suitable Program Dependency Graph Metrics for Defect Prediction Models

The results of RFECV show a median of four optimal features per model, with a mean MCC and standard deviation of 0.64 and 0.31, respectively. However, the lower number of optimal features suggests that most of them are redundant and decrease the overall performance.

Table 5.1: Features importance and rank

Metric	Occurrences	Rank
maxPdgVertices	57	2.21
verticesCount	50	2.92
edgesToVerticesRatio	42	3.95
edgesCount	41	3.88
globalInput	38	4.79
lackOfCohesion	24	13.58
indirectFanOut	19	17.47
indirectFanIn	9	50.33
directFanOut	7	61.14
directFanIn	3	188.33
globalOutput	2	310.5

Table 5.1 shows a ranked list of the recurring features. As we can see, the most important predictors, those that contribute substantially to prediction, are the first 4 and they are: *maxPdgVertices*, *verticesCount*, *edgesToVerticesRatio*, *edgesCount*. We could also consider the fifth predictor *globalInput* because rank value remains in line with the first 4 and diverges considerably from the remaining ones.

> ***RQ_1 Summary***. *The most occurring predictors are* MAXPDGVERTICES, EDGESCOUNT, EDGESTOVERTICESRATIO, *and* VERTICESCOUNT.

5.3 RQ_2 - In Search of the Best Defect Prediction Model based on Program Dependence Graph Metrics

We experimented with how performance varies by including and excluding normalization or data balancing steps to find the best possible combination. The final result shows that the best combination consists of no data balancing and a min-max transformation. Based on this configuration, five classifiers were trained, *Naive Bayes (NB)*, *Logistic Regression (LR)*, *Decision Tree (DT)*, *Random Forest (RF)* and *Support Vector Machine (SVM)*. First, we analyzed the number of times each evaluated classifier achieved the best performance (i.e., it was the best model in terms of MCC) for a given project. Then, as it was difficult to make assumptions about the underlying distribution with many evaluation measures, we applied a non-parametric test to assess the differences' significance. We established the best model by comparing the MCC of the experimented models through the Wilcoxon's rank test [33], applying the post-hoc Bonferroni's correction [34] to deal with the multiple comparisons that were performed during the validation process. In addition, we also computed the Cohen's δ effect size measure [35] to assess the magnitude of the differences observed. A Cohen's d below 0.2 is considered negligible, between 0.2 and 0.5 it is small, between 0.5 and 0.8 it is medium, and it is large above 0.8.

Table 5.2 shows each evaluated model's occurrences as the best model for any given project in terms of MCC. As can be observed, DECISION TREE and RANDOM FOREST are the classifiers that occur most (47/80 and 46/80), followed by NAIVE

Table 5.2: Number of Times a Model Appears Among the Best-Performing Models

Learning Technique	Occurrences
DT	47
RF	46
NB	22
LR	15
SVM	11

BAYES (22/80), LINEAR REGRESSION (15/80), and SUPPORT VECTOR MACHINE (11/80). It is important to mention that the sum of the occurrences is not equal to 80 as, for some projects, multiple models achieved the same performance.

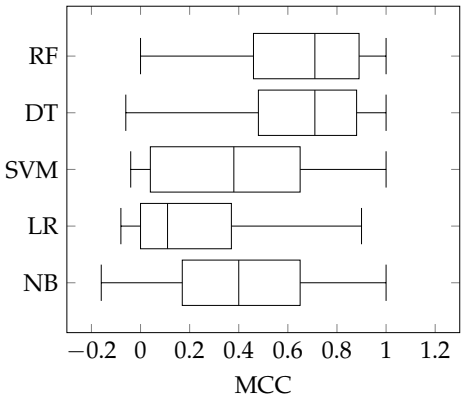

Figure 5.1: Matthews Correlation Coefficient of each Learning Technique

Figure 5.1 shows that the difference among the learning techniques in terms of mean MCC are in most cases very high. The average MCC ranges from 0.25 for the worst-performing technique, namely LINEAR REGRESSION, to above 0.64 and 0.63 for the most-performing techniques, namely RANDOM FOREST and DECISION TREE.

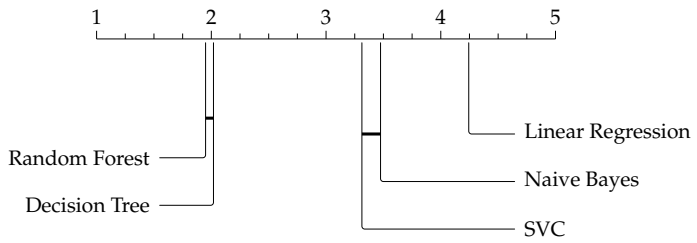

Figure 5.2: Nemenyi's Classifiers Diagram

RQ_2 Summary. *The models trained using Random Forest and Decision Tree perform statistically better than those relying on the remaining classifiers. The difference is statistically different with large effect size.*

Figure 5.2 shows the pairwise comparisons between classifiers. The performance difference between RANDOM FOREST and DECISION TREE is not significant and also between SUPPORT VECTOR MACHINE and NAIVE BAYES.

5.4 RQ_3 - Complementarity between the PDG Metrics-based Model and the Baselines

Upon addressing $\mathbf{RQ_2}$ and assessing the performance of the defect prediction model based on PDG metrics, we compared it with the existing baseline developed by Dalla Palma et al. [20]. Specifically, we ran the baseline and conducted a complementarity analysis to understand the overlap with the PDG-based model. Given the two prediction models, m_i and m_j, we computed (1) the number of bugs correctly predicted by both m_i and m_j, (2) the number of bugs correctly predicted by m_j only 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.

RQ_3 Summary. *PDG metrics improve the number of bugs correctly predicted by the baseline model based on Delta metrics alone by 23.8%. PDG metrics improve the number of bugs correctly predicted by the baseline model based on Process metrics alone by 21.13%. PDG metrics improve the number of bugs correctly predicted by the baseline model based on ICO metrics alone by 1.6%.*

PDG metrics-based models correctly predicted the number of bugs over 20% more than Delta and Process metrics-based models.

Table 5.3: Complementarity between the PDG metrics-based model and the baseline model

	A ∩ B	A \ B	B \ A	Missed by both
PDG - Delta	66.50%	23.80%	4.55%	5.15%
PDG - Process	69.16%	21.13%	4.75%	4.95%
PDG - ICO	88.70%	1.60%	6.92%	2.79%

5.5 RQ_4 - On the Performance of the Baselines and a Novel “Hybrid" Model

In RQ_4, we combined the four metrics sets to construct 15 different hybrid defect prediction models. We employed the same evaluation metrics as RQ_2, i.e., precision, recall, F-Measure, MCC, and AUC-PR. We used the Nemenyi test, a post-hoc test intended to find the groups of data that differ after a global statistical test. As for the statistical test, we used the Friedman test. The procedure involves ranking each row together and then considering the values of ranks by columns.

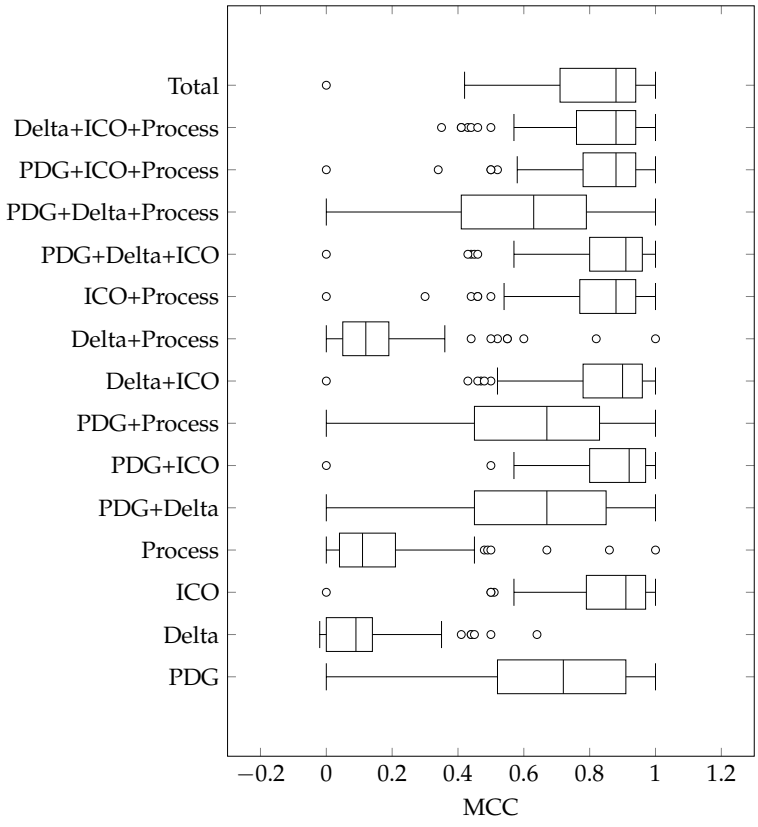

Figure 5.3: Matthews Correlation Coefficient of each Set of Metrics

Figure 5.3 shows that the gap between the model's performance featuring ICO and those relying on PDG, Process, or Delta metrics is evident. However, more sets of metrics produce similar results. PDG metrics and PDG+ICO are not statistically different. Adding PDG metrics to ICO metrics does not improve the model's performance.

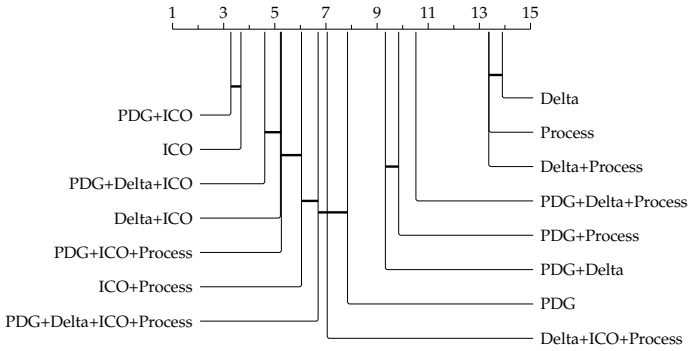

Figure 5.4: Nemenyi post-hoc Critical Distance Diagram

Figure 5.4 shows the metrics compared and grouped into five sets of metrics. Each group contains the combinations of metrics that have a non-statistically significant performance difference. On the one hand, we note that PDG metrics can improve the performance of Delta and Process metrics. On the other hand, the metrics based on the program dependence graph have negligible effects on models employing ICO metrics. In particular, PDG metrics combined with ICO metrics do not improve the performance that ICO metrics alone have. Instead, combined with either Delta metrics or Process metrics, they produce better models than those based on Delta and Process metrics. Finally, models based on Delta and Process metrics produce the worst performance, even when combined.

> ***RQ_4 Summary***. *PDG metrics can improve the performance of Delta and Process metrics. However, such metrics have negligible effects on models employing ICO metrics.*

CHAPTER 6

Discussion, Limitations, and Implications

In RQ1, we observed that the top-4 predictors (in terms of occurrences among the most critical features resulting from the recursive feature elimination) include maxPdgVertices, verticesCount, edgesToVerticesRatio, and edgesCount. All four predictors similarly express the concept of size through the number of nodes and edges and their ratio. A high number of edges indicates greater data flow and control flow. This is much more relevant than the concept of direct or indirect modification of a variable and its use in a task. We conjecture that this result is due to the concept of idempotency in Ansible that limits modification. In RQ2, we observed that the performance difference between Naive Bayes and SVC and between Random Forest and Decision Tree is not statistically significant, although the former provided better results most of the time. Random Forest and Decision Tree share a common algorithm in their implementation. Random Forest is a decision tree-based ensemble method, which means it uses decision trees as the main component of its classification process. Hence, depending on the level of model flexibility you desire and the computational resources at your disposal, you can opt for either option interchangeably without significantly negatively affecting the prediction. The results achieved in the context of RQ3 are surprising. For the collected Ansible-based projects, PDG metrics outperform delta and process metrics, although the latter is often more effective when predicting

the failure-proneness of source code instances in traditional Defect Prediction. We conjecture that this result is due to the lower number of infrastructure code changes than application code, limiting the information exploitable by the process metrics. Anyway, PDG metrics were found to be similar to ICO metrics. The first can correctly predict 90.3% of bugs with respect to the 95.6% of the second. The downside of PDG metrics concerns extraction. The construction of a PDG requires that the Ansible project compile correctly. During the life cycle of a project, it may happen that one or more commits make the project uncompilable. As a result, the effort to extract metrics and the possible absence of a set of metrics for each commit may adversely affect their use. In RQ4, we observed that the metrics based on the program dependence graph have negligible effects on models employing the ICO metrics. The performance of PDG metrics is not statistically different from PDG+Delta+ICO, Delta+ICO, PDG+ICO+Process, ICO+Process, Delta+ICO+Process, and Total. So we can consider PDG metrics a good representative for all these sets of metrics. Finally, PDG metrics can improve the performance of Delta and Process metrics.

Implications for researchers. There is still room for further research in this area. Our findings put a baseline to investigate which prediction models should be used based on the characteristics of the software project to analyze (e.g., the number of contributors and commits, LOC, and the ratio of IaC files). This aspect is of particular interest in the context of Cross-Project Defect Prediction, where the lack of historical data forces organizations to use pre-trained models built on similar projects. Further research is needed to understand the relationship between the failure-proneness of Infrastructure-as-Code and the collected metrics. These results can lead to a better understanding of which features to utilize to improve defect prediction of IaC.

Implications for practitioners. Practitioners who still do not use prediction models for IaC can build upon our findings to implement novel models by extracting only subsets of features such as the ones that we showed in RQ1. This aspect will reduce the number of features to collect and speed up the training phase. For each project on which we trained our models, we reported several statistics to allow practitioners to compare their projects with those used in this study.

CHAPTER 7

Threats to Validity

This section discusses the potential threats that may have affected the validity of our empirical study [24].

Threats to construct validity. Threats to *construct* validity concerns potential issues or challenges that can affect the accuracy and appropriateness of the constructs or concepts being studied. The first possible threat concerns the projects we analyzed in our study. We relied on publicly available resources built in the context of previous research [20] that have already been used and validated, making us confident of the reliability of the selected projects. Another threat concerns how we collected the set of PDG metrics. We used the PDG builder that has already been used and validated [8, 21]. We attempted to perform manual investigations on a statistically significant sample of PDG slices to assess the degree of accuracy of the extracted metrics (Section 4.3)—in this way, we were 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

mitigated this threat by considering a comprehensive set of PDG metrics gathered from the literature [22, 23]. Similarly, data balancing is a critical aspect of defect prediction, so we evaluated several over- and under-sampling techniques and how they affect the model's performance.

Threats to external validity. Threats to *external* validity concern the generalization of results. First, we analyzed 139 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 learned features that characterize failure-prone IaC scripts from the individual projects considered. Projects with a small number of defective instances were 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 evaluated five classifiers widely used in previous studies on bug prediction (e.g., [37, 38, 20]).

Threats to conclusion validity. Concerning the relationship between treatment and outcome, we exploited a set of widely used metrics to evaluate the performance of defect prediction techniques (i.e., precision, recall, F-measure, MCC, AUC-PR) [20, 12, 13]. In addition, we used appropriate statistical tests, i.e., the Wilcoxon Test and the Cohen's Delta, which allowed us to support our findings and address our hypothesis. When assessing the contribution of the features to use in our approach, we relied on the *Recursive Feature Elimination* algorithm [39], which the research community has used for the same purpose [40, 20]. Furthermore, since we exploited 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 used 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 leveraged the SZZ algorithm and committed messages indicating defect-fixing ac-

tivities 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.

CHAPTER 8

Conclusion

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, to improve the early detection of defects in IaC scripts. We started working toward this goal by computing the PDG metrics on a set of 139 Ansible projects. The PDG was extracted using the PDGs proposed by Opdebeeck et al. [8, 21]. Then we experimented with multiple machine learning classifiers. We included *Naive Bayes* [25] and *Logistic Regression* as classifiers that do not require much training data and *Decision Tree* [27], *Random Forest* [28], and *Support Vector Machine* [29], which are more flexible and powerful classifiers. Finally, we conducted many experiments that allowed us to answer the research questions. The PDG metrics that maximize the performance of defect prediction models are *maxPDGVertices*, *verticesCount*, *edgesToVerticesRatio* and *edgesCount* with a mean rank value of 2.21, 2.92, 3.65 and 3.88 respectively. The Program Dependence Graph metrics-based models trained using *Random Forest* and *Decision Tree* performed statistically better than those relying on the remaining classifiers. Program Dependence Graph metrics improved the number of bugs correctly predicted by 23.8% over Delta metrics alone, 21.13% over Process metrics alone, and 1.6% over ICO metrics alone. Finally, PDG metrics can improve the performance of Delta and Process metrics. However, such metrics have negligible effects on models

employing ICO metrics.

In summary, this thesis has explored the realm of defect prediction using Program Dependence Graph metrics, uncovering their significance in enhancing software quality and providing a promising avenue for future research and practical application in software development.

Bibliography

[1] C. Ebert, S. Technologyof, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *Ieee Software*, vol. 33, no. 3, pp. 94–100, 2016. (Citato a pagina 1)

[2] K. Morris, *Infrastructure as code*. O'Reilly Media, 2020. (Citato a pagina 1)

[3] L. Hochstein and R. Moser, *Ansible: Up and Running: Automating configuration management and deployment the easy way*. " O'Reilly Media, Inc.", 2017. (Citato a pagina 1)

[4] J. Ewart, M. Marschall, and E. Waud, *Chef: Powerful Infrastructure Automation*. Packt Publishing Ltd, 2017. (Citato a pagina 1)

[5] J. Loope, *Managing infrastructure with puppet: configuration management at scale*. " O'Reilly Media, Inc.", 2011. (Citato a pagina 1)

[6] 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)*. IEEE, 2019, pp. 580–589. (Citato a pagina 1)

[7] Y. Kurniawan, *Ansible for AWS*. Leanpub, 2016. (Citato a pagina 1)

[8] 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, 2022, pp. 61–72. (Citato alle pagine 1, 11, 12, 17, 24, 39 e 42)

[9] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 189–200. (Citato a pagina 1)

[10] 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)*. IEEE, 2018, pp. 220–228. (Citato a pagina 1)

[11] 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)*. IEEE, 2018, pp. 164–174. (Citato a pagina 1)

[12] 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)*. IEEE, 2018, pp. 34–45. (Citato alle pagine 1 e 40)

[13] ——, "Source code properties of defective infrastructure as code scripts," *Information and Software Technology*, vol. 112, pp. 148–163, 2019. (Citato alle pagine 1 e 40)

[14] A. Rahman, E. Farhana, and L. Williams, "The 'as code' activities: Development anti-patterns for infrastructure as code," *Empirical Software Engineering*, vol. 25, pp. 3430–3467, 2020. (Citato a pagina 1)

[15] 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*, 2020, pp. 752–764. (Citato a pagina 1)

[16] 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)*. IEEE, 2019, pp. 164–175. (Citato a pagina 1)

[17] 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*, vol. 16, no. 3, pp. 291–315, 2020. (Citato a pagina 1)

[18] 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*. IEEE, 2011, pp. 85–95. (Citato a pagina 1)

[19] R. S. Wahono, "A systematic literature review of software defect prediction," *Journal of software engineering*, vol. 1, no. 1, pp. 1–16, 2015. (Citato a pagina 1)

[20] 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*, vol. 48, no. 6, pp. 2086–2104, 2021. (Citato alle pagine 2, 22, 24, 25, 28, 29, 30, 34, 39 e 40)

[21] 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)*, 2023, pp. 534–545. (Citato alle pagine 11, 17, 24, 39 e 42)

[22] 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)*, 2020, pp. 411–422. (Citato alle pagine 11, 24 e 40)

[23] 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*, 2006, pp. 31–42. (Citato alle pagine 11, 24 e 40)

[24] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012. (Citato alle pagine 22 e 39)

[25] R. O. Duda, P. E. Hart *et al.*, *Pattern classification and scene analysis*, ser. A Wiley-Interscience publication. Wiley, 1973. (Citato alle pagine 24 e 42)

[26] M. P. LaValley, "Logistic regression," *Circulation*, vol. 117, no. 18, pp. 2395–2399, 2008. (Citato a pagina 24)

[27] Y. Freund and L. Mason, "The alternating decision tree learning algorithm," in *icml*, vol. 99. Citeseer, 1999, pp. 124–133. (Citato alle pagine 24 e 42)

[28] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001. (Citato alle pagine 24 e 42)

[29] W. S. Noble, "What is a support vector machine?" *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006. (Citato alle pagine 24 e 42)

[30] 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*, vol. 6, no. 1, pp. 20–29, 2004. (Citato a pagina 25)

[31] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of machine learning research*, vol. 13, no. 2, 2012. (Citato a pagina 26)

[32] 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*, vol. 25, pp. 4805–4830, 2020. (Citato a pagina 26)

[33] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945. (Citato alle pagine 28 e 32)

[34] M. A. Napierala, "What is the bonferroni correction?" *Aaos Now*, pp. 40–41, 2012. (Citato alle pagine 28 e 32)

[35] J. Cohen, "The effect size index: d. statinformation and software technologyical power analysis for the behavioral sciences," *Abingdon-on-Thames: Routledge Academic*, 1988. (Citato alle pagine 28 e 32)

[36] J. Yao and M. Shepperd, "The impact of using biased performance metrics on software defect prediction research," *Information and Software Technology*, vol. 139, p. 106664, 2021. (Citato a pagina 28)

[37] 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*, vol. 44, no. 1, pp. 5–24, 2017. (Citato a pagina 40)

[38] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: do different classifiers find the same defects?" *Software Quality Journal*, vol. 26, pp. 525–552, 2018. (Citato a pagina 40)

[39] X.-w. Chen and J. C. Jeong, "Enhanced recursive feature elimination," in *Sixth International Conference on Machine Learning and Applications (ICMLA 2007)*. IEEE, 2007, pp. 429–435. (Citato a pagina 40)

[40] 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)*. IEEE, 2020, pp. 1–6. (Citato a pagina 40)

Appendix

Table 1: Statistical Comparison of Mean MCC Among Learning Techniques. Values below the diagonal are the differences between pairs of techniques. A negative value means that the model in the row performed worse than the one in the column. Values above the diagonal are the effect size.

	NB	**LR**	**SVM**	**DT**	**RF**
NB	-	Medium	Negligible	Medium	Medium
LR	-0.17	-	Small	Large	Large
SVM	-0.01	0.16	-	Medium	Medium
DT	0.21	0.38	0.22	-	Negligible
RF	0.22	0.39	0.23	0.01	-

Table 2: Nemenyi post-hoc test. In the context of a Nemenyi test, a p-value of 0.90 or 0.83 in a pairwise comparison indicates that there is no statistically significant difference between the groups being compared.

	NB	**LR**	**SVM**	**DT**	**RF**
NB	1.00	0.00	0.83	0.00	0.00
LR	0.00	1.00	0.00	0.00	0.00
SVM	0.83	0.00	1.00	0.00	0.00
DT	0.00	0.00	0.00	1.00	0.90
RF	0.00	0.00	0.00	0.90	1.00

Table 3: MCC of each Defect Prediction Model based on PDG Metrics per Project

Project	NB	LR	SVC	DT	RF
PyratLabs/ansible-role-k3s	0.19	0.14	0.23	0.34	0.34
anthcourtney/ansible-role-cis-amazon-linux	0.00	0.00	0.00	0.41	0.41
ansible-ThoTeam/nexus3-oss	0.23	0.13	0.51	0.71	0.70
automium/service-kubernetes	0.21	0.00	0.01	0.48	0.47
oVirt/ovirt-ansible-hosted-engine-setup	0.04	0.00	0.10	0.78	0.78
riemers/ansible-gitlab-runner	0.10	0.03	0.23	0.61	0.61
ansistrano/deploy	0.10	0.31	0.48	0.49	0.48
elastic/ansible-elasticsearch	0.29	0.21	0.55	0.78	0.79
sensu/sensu-ansible	0.02	0.00	0.03	0.07	0.07
cloudalchemy/ansible-grafana	0.21	0.24	0.34	0.72	0.71
openstack/openstack-ansible-os_nova	0.22	0.09	0.36	0.60	0.55
CSCfi/ansible-role-slurm	0.60	0.15	0.66	0.92	0.93
openwisp/ansible-openwisp2	0.40	0.33	0.57	0.69	0.69
cloudalchemy/ansible-prometheus	0.59	0.66	0.75	0.91	0.90
ANXS/postgresql	0.17	0.10	0.31	0.83	0.83
openstack/openstack-ansible-rabbitmq_server	0.21	0.05	0.13	0.7	0.73
galaxyproject/ansible-galaxy	0.47	0.31	0.37	0.67	0.70
Oefenweb/ansible-percona-server	0.00	0.04	0.39	0.91	0.91
UnderGreen/ansible-role-mongodb	0.56	0.68	0.71	0.77	0.75
ansible-community/ansible-nomad	0.64	0.30	0.57	0.89	0.89
openstack/openstack-ansible-os_neutron	0.13	0.00	0.00	0.79	0.79
AlbanAndrieu/ansible-jenkins-slave	0.14	0.24	0.18	0.29	0.38
CoffeeITWorks/ansible_burp2_server	0.42	0.45	0.53	0.74	0.72
DataDog/ansible-datadog	0.55	0.47	0.50	0.73	0.80
oVirt/ovirt-ansible-disaster-recovery	0.20	0.00	0.00	0.73	0.72
stackbuilders/sb-debian-base	0.50	0.00	0.26	0.29	0.26
evrardjp/ansible-keepalived	0.82	0.81	0.85	0.93	0.92
cloudalchemy/ansible-node-exporter	0.76	0.60	0.77	0.86	0.85
HanXHX/ansible-nginx	0.18	0.27	0.46	0.65	0.62
viasite-ansible/ansible-role-zsh	0.26	0.52	0.56	0.81	0.78
lae/ansible-role-proxmox	0.53	0.07	0.35	0.56	0.62

HanXHX/ansible-debian-bootstrap	0.70	0.00	0.79	0.76	0.78
nusenu/ansible-relayor	0.64	0.00	0.48	0.86	0.85
DavidWittman/ansible-redis	0.66	0.00	0.59	0.74	0.75
cloudalchemy/ansible-alertmanager	0.46	0.30	0.49	0.63	0.64
aalaesar/install_nextcloud	0.28	0.39	0.38	0.48	0.41
Oefenweb/ansible-postfix	0.88	0.90	0.94	0.97	0.97
caktus/tequila-django	0.68	0.37	0.74	0.91	0.94
ansible-community/ansible-vault	0.56	0.22	0.56	0.85	0.89
florianutz/Ubuntu1804-CIS	0.25	0.29	0.44	0.65	0.65
dj-wasabi/ansible-telegraf	0.59	0.49	0.58	0.53	0.58
rvm/rvm1-ansible	0.04	0.09	0.30	0.54	0.59
cloudalchemy/ansible-blackbox-exporter	1.00	1.00	1.00	1.00	1.00
tulibraries/ansible-role-airflow	0.07	0.00	0.00	0.00	0.00
idealista/mysql_role	1.00	0.00	0.12	1.00	1.00
Graylog2/graylog-ansible-role	0.20	0.00	0.00	0.00	0.19
mrlesmithjr/ansible-netdata	0.28	0.00	0.28	0.76	0.59
willshersystems/ansible-sshd	0.93	0.89	0.93	1.00	0.98
idealista/consul_role	0.86	0.00	0.86	0.86	0.86
ansible-lockdown/RHEL7-STIG	0.42	0.10	0.57	0.70	0.69
stone-payments/ansible-rabbitmq	1.00	0.00	0.17	1.00	1.00
naftulikay/ansible-role-degoss	0.31	0.31	0.31	0.53	0.53
CSCfi/ansible-role-users	1.00	1.00	1.00	1.00	1.00
mrlesmithjr/ansible-manage-lvm	0.09	0.00	0.12	0.13	0.13
cloudalchemy/ansible-pushgateway	1.00	0.00	0.00	0.95	1.00
antoiner77/caddy-ansible	0.60	0.56	0.60	0.53	0.57
idealista/nexus-role	0.13	0.00	0.00	0.00	0.00
Oefenweb/ansible-supervisor	0.71	0.77	0.65	0.88	0.88
kibatic/ansible-traefik	0.31	0.12	0.04	0.35	0.38
arillso/ansible.logrotate	0.04	0.00	0.00	0.17	0.08
m4rcu5nl/ansible-role-zerotier	0.33	0.33	1.00	1.00	1.00
fgci-org/ansible-role-cuda	0.55	0.00	0.00	0.00	0.33
dokku/ansible-dokku	0.65	1.00	0.91	0.84	0.84
linux-system-roles/storage	0.00	0.00	0.00	0.00	0.00

florianutz/Ubuntu1604-CIS	0.41	0.05	0.41	0.57	0.57
idealista/java_role	0.71	0.00	0.71	0.94	1.00
diodonfrost/ansible-role-mariadb	1.00	0.00	0.00	1.00	1.00
newrelic/infrastructure-agent-ansible	0.71	0.95	0.88	0.95	0.95
elastic/ansible-beats	0.45	0.15	0.00	0.50	0.43
wcm-io-devops/ansible-conga-ansible-controlhost	0.00	0.00	0.0	0.53	0.68
stackhpc/ansible-role-os-images	1.00	1.00	1.00	1.00	1.00
infOpen/ansible-role-docker	0.00	0.00	0.00	0.00	0.00
stackhpc/ansible-role-libvirt-vm	0.22	0.00	0.00	0.00	0.00
nwoetzel/ansible-role-eclipse	0.00	0.00	0.00	0.67	0.67
mimacom/ansible-role-bamboo	0.50	0.50	1.00	0.83	0.83
AlexeySetevoi/ansible-clickhouse	1.00	0.00	1.00	1.00	1.00
hadret/ansible-role-containers	0.12	0.12	0.12	0.12	0.12
infOpen/ansible-role-fail2ban	1.00	1.00	1.00	1.00	1.00
cloudalchemy/ansible-smokeping_prober	0.00	0.00	0.00	0.00	0.00
galaxyproject/ansible-galaxy-tools	0.00	0.00	0.00	0.00	0.00
Average mcc	**0.42**	**0.25**	**0.41**	**0.63**	**0.64**
Average rank	**3.47**	**4.24**	**3.31**	**2.01**	**1.95**

Table 4: Performance Statistics of Random Forest Across the 80 Repositories

	count	mean	std	min	25%	50%	75%	max
Precision	80	0.70	0.30	0.00	0.60	0.79	0.92	1.00
Recall	80	0.71	0.31	0.00	0.55	0.83	0.97	1.00
F1	80	0.68	0.30	0.00	0.57	0.77	0.90	1.00
Mcc	80	0.64	0.31	0.00	0.46	0.71	0.89	1.00
Auc-pr	80	0.79	0.23	0.00	0.67	0.86	0.96	1.00

Table 5: Performance Statistics of Decision Tree Across the 80 Repositories

	count	mean	std	min	25%	50%	75%	max
Precision	80	0.70	0.31	0.00	0.61	0.81	0.92	1.00
Recall	80	0.69	0.33	0.00	0.56	0.81	0.96	1.00
F1	80	0.67	0.31	0.00	0.53	0.77	0.90	1.00
Mcc	80	0.63	0.32	0.00	0.48	0.71	0.88	1.00
Auc-pr	80	0.79	0.21	0.00	0.67	0.86	0.94	1.00

Table 6: Nemenyi pairwise comparisons test between set of metrics. Comparisons of Delta with Delta+PDG, Process with Process+PDG, and Delta+Process with Delta+Process+PDG metrics show a statistically significant difference in performance. Sets of metrics that additionally contain PDG metrics perform better than sets that do not contain PDG metrics. On the other hand, sets of metrics that contain ICO metrics, e.g., ICO, ICO+Delta, ICO+Process, and ICO+Delta+Process, have performances that are not statistically different from their respective sets with PDG metrics added.

	δ	*ICO*	*P*	*δ+ICO*	*δ+P*	*ICO+P*	*δ+ICO+P*
PDG+δ	**0,00**	0,00	0,00	0,00	0,00	0,00	0,09
PDG+ICO	0,00	**0,90**	0,00	0,28	0,00	0,01	0,00
PDG+P	0,00	0,00	**0,00**	0,00	0,00	0,00	0,01
PDG+δ+ICO	0,00	0,90	0,00	**0,90**	0,00	0,75	0,04
PDG+δ+P	0,00	0,00	0,00	0,00	**0,00**	0,00	0,00
PDG+ICO+P	0,00	0,62	0,00	0,90	0,00	**0,90**	0,42
PDG+δ+ICO+P	0,00	0,00	0,00	0,71	0,00	0,90	**0,90**

δ = Delta Metrics *P = Process Metrics*

ICO = IaC-Oriented Metrics *PDG = PDG Metrics*

Acknowledgements

I would like to express my deep gratitude to all those who helped make this thesis possible. Their knowledge, support, and inspiration have been fundamental to my academic and personal journey. Many thanks to everyone who shared this extraordinary adventure with me. In a special way, I would like to extend sincere thanks to my mother and father. Your love, constant support, and trust in me have been the main driving force behind my achievement. Without you, it would not have been possible. Thank you from the bottom of my heart for everything you have done for me.