CATCHING BUGS IN THE CLOUD

PREDICTIVE MAINTENANCE OF INFRASTRUCTURE-AS-CODE VIA DEFECT PREDICTION AND CODE SMELL DETECTION TO CONTINUOUSLY IMPROVE QUALITY

Stefano DALLA PALMA

CATCHING BUGS IN THE CLOUD

PREDICTIVE MAINTENANCE OF INFRASTRUCTURE-AS-CODE VIA DEFECT PREDICTION AND CODE SMELL DETECTION TO CONTINUOUSLY IMPROVE QUALITY

Proefschrift

Proefschrift ter verkrijging van de graad van doctor aan TILBURG UNIVERSITY op gezag van de rector magnificus, prof. dr. W.B.H.J. van de Donk, in het openbaar te verdedigen ten overstaan van een door het college voor promoties aangewezen commissie in de Portrettenzaal van de Universiteit op

dindsdag 28 maart 2023
om 16.00 uur

door

Stefano DALLA PALMA

geboren te Venafro, Italië

European Commission | Horizon 2020 European Union funding for Research & Innovation

Dit proefschrift is uitgevoerd met de steun van het Horizon 2020 onderzoeks- en innovatieprogramma van de Europese Unie – grant No. 825040 (RADON H2020).

Naam drukkerij: Gildeprint

Catching Bugs in the Cloud

Predictive Maintenance of Infrastructure-as-Code via Defect Prediction and Code Smell Detection to Continuously Improve Quality

Dissertation

Dissertation to obtain the degree of doctor from Tilburg University on the authority of the Rector Magnificus prof W.B.H.J. van de Donk, according to the decision of the Doctorate Board to be defended in public on

Tuesday, March 28, 2023
at 4.00 pm

by

Stefano Dalla Palma

born in Venafro, Italy

European Commission | Horizon 2020 European Union funding for Research & Innovation

This dissertation has been carried out under the support of the European Union's Horizon 2020 research and innovation program – grant No. 825040 (RADON H2020).

Printed by Gildeprint

About the Cover Illustration

The figure[1] on the cover of *Catching Bugs in the Cloud* – a frog intending on catching a moth – has a two-fold meaning.

First, intuitively, frogs eat insects such as flies and moths. They use long tongues and sticky saliva to catch preys that pass them by. Similarly, the tool developed in this thesis acts as a frog, a “guardian” catching the bugs hidden in Cloud infrastructures.

The second reason is connected to the city I have been living in during my Ph.D., Den Bosch. Every year during Carnival, all inhabitants of the Dutch city of Den Bosch, or *Oeteldonk* during Carnival, gather to party as *Oeteldonkers* for three whole days. In fact, every city in the Netherlands gets an alternative name during Carnival. These names often have something to do with the dialect of that region or with a historic event that is linked to the city. So, every year Den Bosch becomes *Oeteldonk*, which translates to *Frog Hill*, from “oetel” meaning frog and “donk” meaning sandy hill.

Oeteldonk is based on a higher place than its surroundings. In medieval times, this placement of the city had strategic value because you could see your enemies approaching from atop the hill. Also, Oeteldonk’s was the only dry sand dune in the scenery; it was surrounded by swamps purposely created to shield the city from intruders and attackers. Legend says that these swamps were filled with frogs. The frog thus became the most important symbol of Oeteldonk although it has not been proven that those swamps were truly filled with frogs. This symbol constructs meaning to Oeteldonk: for all the Oeteldonkers the frog – a mythical symbol of the city – is connected to Carnival, an intense celebration for the city.[2]

[1] Adapted from https://pin.it/4fJYAuZ

[2] https://www.diggitmagazine.com/articles/exclusiveness-oeteldonkers

ABSTRACT

Infrastructure-as-Code (IaC) – executable code for provisioning and configuring software-defined infrastructures and orchestrating services in cloud applications – became a crucial practice to automatically drive the operationalization of large-scale software systems by ensuring a fully documented, versioned infrastructure and repeatable routines for service provisioning through executable scripts.

Simultaneously, IaC emerged as the de-facto challenge for software maintenance and evolution to cope with software infrastructures that are constantly evolving and growing in size and complexity. Indeed, IaC has the same potential problems as traditional code, such as lousy coding practices known as *code smells* and deficiencies hindering correct functioning, commonly referred to as *defects*. Not only do they reduce infrastructure code quality, but they can also result in costly outages and high-impact business and society. For example, think of the recent faulty configuration change on the backbone routers coordinating network traffic between Facebook's data centers, which cost Facebook nearly $100 million in revenue and profoundly impacted billions of people and businesses worldwide.

Software Defect Prediction helps mitigate these risks by identifying the parts of the system that are more prone to fail before defects are discovered to effectively optimize the allocation of limited resources for testing and maintenance. Therefore, high-quality, low-cost, maintainable software can be deployed in time, resources and budget. However, when this research began in 2019, the IaC state-of-the-practice lacked quality management techniques and tools such as code smell detection and defect prediction. As a consequence, writing bug-free IaC was primarily left to the experience of developers and operators. Ever since, although IaC research and tools have received increasing attention, there has been a need for more tools to support developers and operators develop and maintain high-quality IaC. Such tools will help cope with the continuous and rapid changes that software-defined infrastructures are facing nowadays.

This thesis tackles these aspects and proposes to advance the state-of-the-art and state-of-the-practice by providing methods and tools for defect prediction and code smell detection for IaC using Machine Learning and their evaluation by large-scale empirical studies. Also, this work was part of the European Union's Horizon-2020 project called RADON, aimed at pursuing a broader adoption of serverless computing technologies within the European software industry (`https://radon-h2020.eu/`). One of its key pillars is to raise companies' perception of the value of high-quality, bug-free IaC and ease its development and quality assurance, to which this thesis contributes by developing and presenting DEFUSE, a modular tool for defect prediction.

DEFUSE is instantiated for Infrastructure-as-Code by focusing on Ansible, one of the most used configuration management technology, and the technology-agnostic OASIS standard for IaC, TOSCA. Nevertheless, although implemented for infrastructure code, DEFUSE was designed to potentially support any infrastructure and application code.

The tool is open-source on GitHub (*radon-h2020/radon-defuse*) and a video tutorial is available on YouTube (`https://youtu.be/37mmLdCX3jU`).

Building DEFUSE required following an iterative process consisting of requirements and data elicitation with the industry partners in RADON, design, implementation, and validation. Attempts to address the challenges and limitations mentioned above led to the following high-level research questions, namely

- *What code metrics can be employed to characterize the quality of, and identify problems in, Infrastructure-as-Code?*
- *Can a Machine Learning approach be employed during IaC Quality Assurance to accurately detect defects and code smells using the elicited metrics?*

Answering the research questions above resulted in the findings and contributions reported in this thesis. First, it proposes a catalog of 46 metrics to evaluate the different aspects of IaC and understand what code metrics can be employed to characterize failure-prone or lousy Infrastructure-as-Code. Then, it proposes a fully integrated Machine Learning framework for infrastructure code quality and correctness that allows for repository crawling, metrics collection based on the devised catalog, model building, and evaluation. Afterward, the framework is evaluated via empirical study.

Findings show that (i) within-project defect prediction of IaC based on Machine Learning can generally reach high performance and (ii) product metrics can be more effective predictors than process metrics for IaC. Finally, this thesis concludes with lessons learned and open issues that need to be addressed by the research community in the future, as well as an overview of future directions.

List of Publications

7. **S. Dalla Palma**, C. van Asseldonk, G. Catolino, D. Nucci, F. Palomba, and D. Tamburri. ”Through the looking-glass ...” An empirical study on blob infrastructure blueprints in the Topology and Orchestration Specification for Cloud Applications. *Journal of Software: Evolution and Process*, 2022.

6. **S. Dalla Palma**, D. Nucci, and D. Tamburri. Defuse: A Commit Annotator and Model Builder for Software Defect Prediction. *In Proceedings of the 38th IEEE International Conference on Software Maintenance and Evolution*, 2022.

5. **S. Dalla Palma**, G. Catolino, D. Di Nucci, D. A. Tamburri, and W.J. Van Den Heuvel. Go Serverless With RADON! A practical DevOps Experience report. *IEEE Software*, 2021.

4. **S. Dalla Palma**, D. Di Nucci, F. Palomba, and D. Tamburri. Within-project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics. *Transactions on Software Engineering*, 2021.

3. **S. Dalla Palma**, M. Garriga, D. Di Nucci, D. A. Tamburri, and W.J. van den Heuvel. DevOps and Quality Management in Serverless Computing: The RADON Approach. *Proceedings of the 8th European Conference On Service-Oriented And Cloud Computing (ESOCC)*, 2020.

2. **S. Dalla Palma**, D. Di Nucci, and D. A. Tamburri. AnsibleMetrics: A Python library for measuring Infrastructure-as-Code blueprints in Ansible. *SoftwareX*, 2020.

1. **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*, 2020.

CONTENTS

LIST OF ABBREVIATIONS

AUC-PR	Area Under the Curve - Precision-Recall	48
CART	Classification And Regression Trees	48
CD	Continuous Delivery	3
CI	Continuous Integration	3
IaC	Infrastructure-as-Code	2
ICO	Infrastructure-as-Code Oriented	41
LR	Logistic Regression	48
MCC	Matthews Correlation Coefficient	48
ML	Machine Learning	6
MSR	Mining software repositories	111
NB	Naive Bayes	48
RF	Random Forest	48
SVM	Support Vector Machine	48
TOSCA	Topology and Orchestration Specification for Cloud Applications	2
VIF	Variable Inflation Factor	60

1

Introduction

1.1. Application Context

The continuous and increasing demand for fast and high-quality software services and their orchestration introduced new challenges and opportunities. The creation of software and its deployment has become more accessible, affordable, and rapid, as the developers can focus on the business logic while delegating the infrastructure management to those having solid expertise in the field. As a result, the time between committing code and deploying it to production has shortened (Sadiku et al. 2014).

Despite that, infrastructure must cope with continuous and rapid changes common in established software development practices, like Agile development. Those changes may introduce potential problems and, ultimately, lead to undesirable infrastructure failures; Table 1.1 gives an overview of some significant infrastructure failures reported in the last couple of years. New methods and tools are needed to help developers and operators define high-quality, bug-free software infrastructures to prevent those failures.

1.1.1. From on-premise to cloud-based infrastructure

Not long ago, companies used to provision and managed their software on their in-house server and computing infrastructure. Besides the implied costs involved in buying and maintaining machines or hiring IT personnel, on-premise provisioning – the process of setting up IT infrastructure to allow it to provide new services to its users – is manual-intensive labor of system administrators that is typically slow, time-consuming, effort-prone, and often error-prone (Morris 2016). Think of the physical setup of the hardware, installation, and configuration of the operating system and related software, network, storage, and more. Hence, those companies started outsourcing some responsibilities, such as managing and maintaining the infrastructure.

The advent of the Cloud pushed this outsourcing even more. Through virtualization and containerization, cloud-based infrastructure eliminated the problem of physical hardware management, enabling developers to delegate infrastructure management to consolidated companies, provision their virtual infrastructure or containers on-demand,

Company	Description
Salesforce	In 2019, sales agents and marketers around the world were not able to access their customer information for at least a day (BroadGroup 2019)
Brithish Airways	In 2019, system failures caused more than a 100 flights to be canceled and over 200 to be delayed (TheGuardian 2019)
O2	In 2018, 30 million users were unable to access data services (TheGuardian 2018)
TSB Bank	In 2018, an outage of the new banking platform caused customers to be locked out of their account, the money disappeared from online accounts, and people were able to see other customers' accounts. It cost the bank 330 million British Pounds and made many customers leave (ComputerWeekly.com 2019)
AWS	In 2017, an outage of an AWS data center cluster of 4 hours affected a large part of the internet. The economic loss was estimated at 150 million dollars (DataCenterKnowledge 2017).
HSBC Bank	In 2016, a large outage made the online and mobile banking services unavailable for two days (ComputerWeekly.com 2016)
Starbucks	In 2015, due to a software bug, stores were unable to process payment transactions. Therefore, 60% of the Canadian and US stores were forced to close (GeekWire 2015)

Table 1.1: Noteworthy software failures of the past years.

and focus on business logic. However, on the flip side, they still had to provide their virtualized infrastructure for every new deployment and address untracked environment changes and inconsistencies that might impact deployments.[1]

1.1.2. Infrastructure-as-Code: Opportunities and Challenges

Infrastructure-as-Code

Machine-readable code for provisioning and configuring software-defined infrastructures and orchestrating services in cloud applications (Morris 2016)

Infrastructure-as-Code (IaC) rapidly became a crucial practice to automatically drive the operationalization of large-scale software systems by ensuring a fully documented, versioned infrastructure and repeatable routines for service provisioning through executable scripts often called application blueprints. Please note that, to our knowledge, there is no standard, established definition of an application blueprint. For example, IBM defines it as a "mapping of application components to a resource template, which specifies a collection of resources, typically including agent prototypes; the blueprint provides a pattern for an environment, including which components are deployed on which agent."[2] Cloudify defines a blueprint as "a YAML file with definitions of resources and connections between them, the syntax is based on the Topology and Orchestration Specification for Cloud Applications (TOSCA) specification."[3] In TOSCA, discussed later

[1] `https://www.ibm.com/cloud/learn/infrastructure-as-code`
[2] `https://www.ibm.com/docs/en/urbancode-deploy/7.1.1?topic=deployment-application-blueprints`
[3] `https://docs.cloudify.co/latest/trial_getting_started/introduction_to_blueprints/writing_blueprint/`

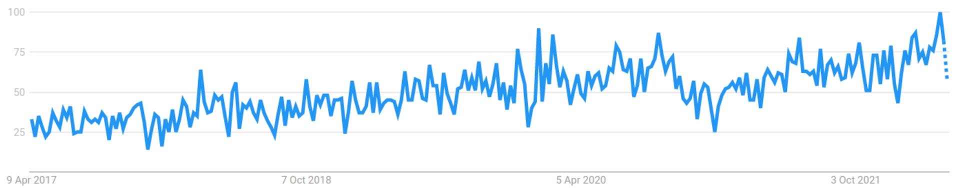


Figure 1.1: Google Search trend for the term “Infrastructure as Code” in the past five years. Numbers represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. A score of 0 means that there was not enough data for this term.

on, an application blueprint is typically used interchangeably to service template, a document written in YAML that adheres to the syntax defined by the specification.

Blueprint

A description of a software-defined infrastructure that can be used as a template for code generators for its automated provisioning, deployment, and run-time management.

Based on those definitions, in the context of this dissertation, we define a blueprint as *“a description of a software-defined infrastructure that can be used as a template for code generators for its automated provisioning, deployment, and run-time management. It specifies infrastructure’s components – i.e., hardware, software, and networks –, configuration, relationships, and capabilities, along with policies for provisioning, deployment, and run-time management.”*

Practitioners consider IaC a fundamental pillar to implementing DevOps practices, which helps them rapidly deliver software and services to end-users (Rahman, Mahdavi-Hezaveh, et al. 2019), as the Google Search trend shows in Figure 1.1 for the past five years. IaC reduces the time, effort, and specialized skills required to provision and scale infrastructure services and improves consistency by reducing ad-hoc configuration changes and updates (Morris 2016). Furthermore, it has substantial benefits on the maintainability and quality of the overall service properties and service-level agreements, e.g., faster closing-the-loop iterations and service evolution cycles, which is critical in the modern IT market that is increasingly focused on the “need for speed”: speed in deployment, faster release-cycles, speed in recovery, and more.

With IaC, infrastructure managers can create scripts that automatically change and maintain the infrastructure using engineering practices proven effective for application development, such as version control systems, automated testing, Continuous Integration (CI), and Continuous Delivery (CD). This automation and accompanying orchestration capabilities provide users with a high degree of self-service to provision resources, connect services, and deploy workloads without direct intervention of the cloud provider.[4]

The resulting paradigm shift in software design, development, and operations management let IaC emerge as the de-facto challenge for software maintenance and evolution of the coming years to cope with IT infrastructures that are constantly evolving and growing in size and complexity. Indeed, IaC has the same potential problems as traditional code, such as lousy coding practices known as code smells and deficiencies hindering correct functioning, commonly referred to as defects. Not only do they reduce the quality of the code underlying the software infrastructure, but they can also result in costly system outages and high-impact business and society.

For example, on October 4th, 2021, a faulty configuration change on the backbone routers coordinating network traffic between Facebook's data centers caused issues that interrupted this communication for more than seven hours.[5] This disruption to network traffic had a cascading effect on how these data centers communicate, bringing Facebook's services to a halt.[6] According to a report produced by Fortune and Snopes, the outage cost Facebook nearly $100 million in revenue and profoundly impacted billions of people and businesses worldwide.[7] Every year, similar examples pop up, with costs varying from $100,000 per hour to $500,000 per hour and even to $1,000,000 per hour for most critical failures.

Although the numbers vary, it became clear that both society and businesses benefit from tools that improve infrastructure code quality. Furthermore, although powerful, combining CI/CD and IaC requires defining many blueprints for instantiating and integrating all the technologies at play within the life-cycle management of cloud applications. These operations emphasize the importance of writing bug-free, high-quality infrastructure code to predicate both "ease of use" (i.e., how easily users can use a product) and "ease of reuse" (i.e., how easily users can reuse existing components). Unfortunately, by the time this research began in 2019, little was known concerning how to best maintain, speedily evolve, and continuously improve the code driving them; yet, it was gaining more traction in different domains (Jarschel 2013; Soldani et al. 2015; Lipton et al. 2018). Besides, state-of-the-practice lacked quality tools for IaC (Guerriero et al. 2019) – a need tackled by this thesis.

1.1.3. Infrastructure-as-Code Quality: Research and Proposition

Although IaC is attracting attention in the scientific community, IaC quality research is in its infancy; hence, only a few research studies have been conducted on its adoption and maintenance. The most relevant that motivated this thesis are Rahman, Mahdavi-Hezaveh, et al. 2019 and Guerriero et al. 2019. The former investigated the current development practices of infrastructure code in the industry and raised the need for more research studies on instruments to support developers and operators when developing and maintaining IaC. The latter analyzed the current research on infrastructure code and

[4] `https://www.frontlettechnologies.com/2022/07/23/an-introduction-to-microsoft-azure-and-cloud-computing/`

[5] `https://techhq.com/2021/10/facebooks-outage-highlights-the-big-problem-with-the-remote-work-policy/`

[6] `https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/`

[7] `https://fortune.com/2021/10/04/facebook-outage-cost-revenue-instagram-whatsapp-not-working-stock/`

raised the need for research studies on defects and security flaws for IaC.

In this research line, to raise companies' perception of the value of high-quality, bug-free IaC and ease its development, a modular tool for defect prediction, called **Defuse**, was developed and presented in this dissertation. This modular characteristic allows Defuse to potentially support any defect prediction exercise, regardless of the code for which defects are detected.

Defect prediction helps software engineers optimize the allocation of limited resources for testing and maintenance by identifying the system components that are more prone to fail before defects are discovered. Therefore, high-quality, low-cost, maintainable software can be deployed in time, resources, and budget (Zhiqiang et al. 2018).

Although it is evident that a tool that correctly detects as many defects as possible is invaluable, this is still far away in terms of feasibility. Therefore, researchers and practitioners split the work and developed smaller software components that tackle a specific problem for a specific coding language. Likewise, Defuse is instantiated for Infrastructure-as-Code by focusing on Ansible, one of the most used configuration management technology in the industry, and TOSCA, the technology-agnostic OASIS standard for IaC.[8]

Ansible is becoming the de-facto reference configuration management technology in the industry due to its simplicity compared with other more complex technologies such as Chef and Puppet. At the same time, TOSCA is increasingly used due to its ability to build upon existing configuration and orchestration languages to improve the readability and portability of configuration files across platforms.

The tool is open-source on GitHub[9] and a video tutorial is available on YouTube.[10]

1.2. Research Statement

Problem

The IaC state-of-the-practice lacks quality management techniques such as code smell detection or defect prediction and tools that help developers and operators write and maintain bug-free, high-quality infrastructure code during Quality Assurance and testing.

"Cloud native technologies allow organizations to build and run scalable applications dynamically through Infrastructure-as-Code (IaC) templates. However, when these templates are not scanned for security issues, they can unnecessarily expose an organization's cloud environment" (PaloaltoNetworks Research 2022). PaloaltoNetworks Research 2022 reported nearly 200,000 insecure IaC templates with high and medium severity vulnerabilities, yet it only takes one such misconfiguration to compromise an entire cloud environment.

[8] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca
[9] https://github.com/radon-h2020/radon-defuse
[10] https://youtu.be/37mmLdCX3jU

When this research began in 2019, bug-free IaC was primarily left to the experience of developers and operators. In traditional software engineering, much research has been devoted to providing quality measures related to technical debt (Münch et al. 2013) and recognizing it using code-smelliness (Tufano et al. 2017) and defect-proneness (D'Ambros et al. 2012a). However, to our knowledge, none of the technical debt management techniques, such as code smell detection or defect prediction and respective tools, have been widely investigated for IaC. Although the problem has recently attracted several scientific works, this research highlights some aspects that might be improved, summarized as follow.

Infrastructure-as-Code Quality. Little is known about how to best maintain, speedily evolve, and continuously improve the code behind the IaC practice in a measurable fashion. In general-purpose programming languages, source code metrics are often computed and analyzed to evaluate the different software quality aspects. However, IaC is domain-specific and traditional metrics may not apply. Therefore, defining source code metrics able to model the quality aspects of IaC may pose the basis for a more extensive evaluation of infrastructure code quality.

Infrastructure-as-Code Defect Prediction. Similarly to other source code artefacts, IaC files may contain defects that can preclude their correct functioning. Several works focus on IaC defect prediction; however, little is known about the role played by classification techniques, source code metrics (a.k.a. product metrics), and process metrics in predicting defective IaC files – for example, process metrics often outperformed source code metrics in traditional defect prediction.

Infrastructure-as-Code Smell Detection. Code smells are broadly researched for traditional source code development, as their detection can enhance software quality. However, in IaC, only a few works exist, focusing on specific technologies and languages, such as Puppet or Ansible. At the same time, there is a lack of knowledge regarding the development behavior for technology-agnostic infrastructure code, such as TOSCA, and how this behavior relates to existing code smells.

In this research, we conjecture that identifying code smells in technology-agnostic infrastructure code (i.e., TOSCA) and related metrics opens up opportunities for building a general, automated service continuity quality model designed explicitly for configuration orchestration languages. Furthermore, analyzing such code smells paves the way to understand how lousy coding practices affect service infrastructure continuity.

Infrastructure-as-Code Tools. Although IaC research and tools are receiving increasing attention, there is a need for more tools to support developers and operators in developing high-quality, bug-free IaC. Furthermore, such tools will help cope with the continuous and rapid changes that software-defined infrastructures face nowadays.

This thesis tackles these aspects by providing methods and tools to help improve the quality of IaC, assisting developers with Machine Learning (ML) techniques that recognize code portions expected to be more prone to defects based on training data from IaC libraries and codebases. This novel area is particularly active and promising for general-purpose languages within the software engineering community. Here, the main challenge for infrastructure code is to address its intrinsic polyglot nature, offering a defect prediction tool which is either *agnostic of IaC technologies* or *specific to address certain IaC defects or anti-patterns*. For example, a technology-agnostic approach could feature Machine Learning, requiring heavy experimentation to train models on diverse languages or using characteristics common to diverse languages. Conversely, specific defects could require feature selection and comparison across a spectrum of Machine Learning predictors to select the best ones for the system and IaC feature at hand, e.g., neural networks, Bayesian networks, random forests, and others.

1.3. Research Goal and Questions

Goal

This thesis proposes to advance the state-of-the-art and state-of-the-practice of IaC quality assurance, focusing in particular on developing and validating tools for code smell detection and defect prediction for IaC using Machine Learning, and their evaluation by large-scale empirical studies.

Attempts to address the challenges and limitations mentioned above led to the following high-level research questions.

RQ$_1$ What code metrics can be employed to characterize the quality of, and identify problems in, Infrastructure-as-Code?

Identifying quality problems in infrastructure code may help in several ways, such as build a general, automated service continuity quality model for configuration orchestration languages, and understand how lousy coding practices affect service infrastructure continuity. However, while quality metrics for traditional code have been widely investigated, these cannot necessarily be transferred, and respective new metrics have to be devised and evaluated for Infrastructure-as-Code.

RQ$_2$ Can a Machine Learning approach be employed during IaC Quality Assurance to accurately detect defects and code smells using the elicited metrics?

Detecting defects and code smells in infrastructure code is crucial for software engineers to focus on critical scripts during quality assurance and allocate effort and resources more efficiently. To achieve this, we aim to develop a practical Machine Learning framework that can be integrated into a software quality pipeline for software-defined infrastructure. The basis for this approach is that we can use predictive automations that can adapt to the complexity of the context of the language under consideration. Traditional rule-based approaches might work, but they only give insights into the individual features being examined and would require a deep understanding of the language, making them less generalizable. Conversely, a machine learner can derive insights from a set of different features and the relationships between them, enabling us to build a more robust and generalizable solution.

Therefore, the research question is designed to investigate the applicability of Machine Learning for infrastructure code quality assurance. In particular, RQ_2 led to more fine-grained research questions regarding the construction of a practical Machine Learning framework:

$RQ_{2.1}$ HOW IS THE PREDICTION PERFORMANCE AFFECTED BY CHOICE OF THE CLASSIFIER SELECTION? Identifying the effect that the choice of Machine Learning classifiers has on the prediction performance may help implement the classification technique that best fits the project's requirements. For example, performance-over-explainability – in which case, for the sake of explainability, a less performant Decision Tree may be preferable to a more performant Support Vector Machine – or vice versa.

$RQ_{2.2}$ HOW IS THE PREDICTION PERFORMANCE AFFECTED BY CHOICE OF THE METRIC SETS? Identifying the effect that the choice of metrics (i.e., code and process metrics and groups thereof) has on the prediction performance helps focus on a subset of features and speed-up feature collection and model training.

$RQ_{2.3}$ WHAT METRICS ARE THE MOST EFFECTIVE IN MAXIMIZING DEFECT PREDICTION PERFORMANCE? Identifying the predictors that highly affect the prediction performance helps further understand and assess the quality of IaC scripts based on their importance for the prediction.

Likewise,

$RQ_{2.4}$ TO WHAT EXTENT CAN STRUCTURAL CODE METRICS DISTINGUISH BETWEEN BLOB – I.E., TOO LARGE AND COMPLEX – AND SOUND BLUEPRINTS? Identifying the structural properties of Blob blueprints will enable their automatic detection and refactoring.

$RQ_{2.5}$ TO WHAT EXTENT CAN METRIC- AND UNSUPERVISED LEARNING-BASED TECHNIQUES DETECT THOSE LOUSY BLUEPRINTS? Metrics-based smell detectors have been extensively used for application code smell detection because of their performance and implementation ease; however, they rely on establishing a suitable detection threshold whose determination is a non-trivial challenge. Conversely, unsupervised learning-based detectors overcome such shortcomings. In addition, they can potentially reduce the effort to identify smelly blueprints and label them in a supervised setting.

$RQ_{2.6}$ WHAT METRICS ARE THE MOST EFFECTIVE TO MAXIMIZE THE PERFORMANCE OF THOSE DETECTORS? Identifying the metrics that highly affect the detection performance helps further understand and assess the quality and smelliness of IaC scripts based on their importance for detection.

These research questions are further detailed in the related chapters; the answer to these questions will be the basis for the development of a practical Machine Learning framework that supports decision-making during infrastructure code quality assurance.

Chapter	Aspect	Publication	Peer-review Venue
3	IaC Quality Metrics	P1	*Journal of Systems and Software*
4, 5	IaC Defect Prediction	P4	*Transactions on Software Engineering*
	Effect of detection techniques		
	Effect of metrics		
	Best predictors		
6	IaC Smell Detection	P7	*Journal of Software: Evolution and Practice*
	Effect of detection technique		
	Effect of metrics		
	Best predictors		
7, A, B	IaC tools		
	AnsibleMetrics	P2	*SoftwareX*
	RepositoryMiner		
	Defuse	P6	*International Conference on Software Maintenance and Evolution*

Table 1.2: Summary of the thesis contribution. Note: the publication id, e.g., P1, refers to the respective publications mentioned in the list of publication attached to the beginning of this disseration.

1.4. RESEARCH CONTRIBUTION

The contribution of this thesis concerns the four aspects introduced in Section 1.2 and are further described below; Table 1.2 reports, for each aspect, the references to papers published or submitted; while Figure 1.2 reports, for each chapter, the research question(s) addressed and a contribution summary.

C_1 Chapter 3 contributes to this thesis with a catalog of 46 metrics to evaluate the different aspects of IaC – the most comprehensive measures set for IaC to date. The metrics target Ansible but are portable to other languages such as Chef and Puppet and offer a general purpose metrics-based approach for IaC quality evaluation and defect prediction.

C_2 Chapter 4 contributes to this thesis by proposing a fully integrated Machine Learning framework for infrastructure code quality and correctness that allows for repository crawling, metrics collection based on the metrics resulting from the previous contribution (C_1), model building, and evaluation. The framework allowed us to gather a comprehensive and meaningful set of data to implement learning-based methods for predicting the failure-proneness of IaC scripts, enabling the following contribution.

C_3 Chapter 5 contributes to this thesis with a large empirical study to compare Machine Learning classifiers and metric sets to detect failure-prone IaC files. Specifically, that contribution provides (i) the empirical evidence that within-project defect prediction of Infrastructure-as-Code based on Machine Learning can generally reach high performance; (ii) the empirical evidence that product metrics can be more effective predictors than process metrics for within-project defect prediction of IaC; (iii) a comprehensive dataset of publicly available failure-prone Ansible files, source code measurements calculated upon these files, and their manual

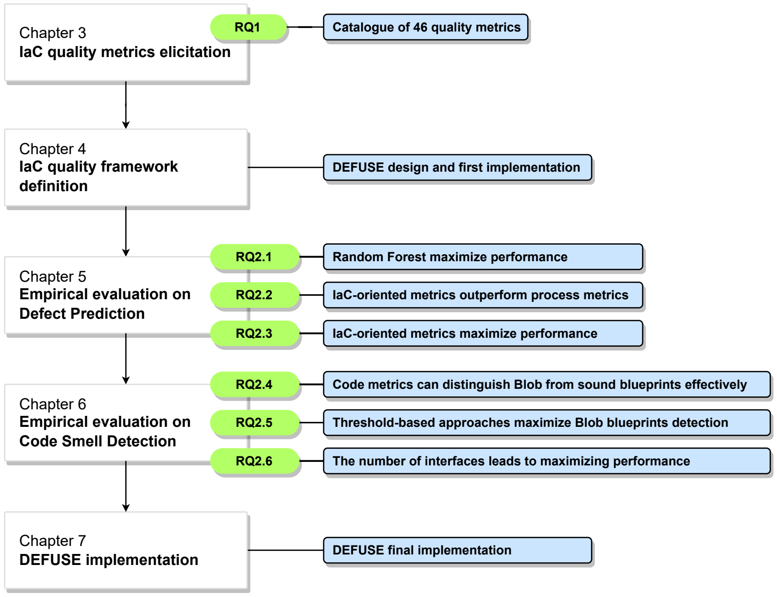

Chapter 3
IaC quality metrics elicitation
RQ1
Catalogue of 46 quality metrics
Chapter 4
IaC quality framework definition
DEFUSE design and first implementation
Chapter 5
Empirical evaluation on Defect Prediction
RQ2.1
Random Forest maximize performance
RQ2.2
IaC-oriented metrics outperform process metrics
RQ2.3
IaC-oriented metrics maximize performance
Chapter 6
Empirical evaluation on Code Smell Detection
RQ2.4
Code metrics can distinguish Blob from sound blueprints effectively
RQ2.5
Threshold-based approaches maximize Blob blueprints detection
RQ2.6
The number of interfaces leads to maximizing performance
Chapter 7
DEFUSE implementation
DEFUSE final implementation

Figure 1.2: Thesis outline and mapping between research question and contribution. While chapters 3, 5, and 6 address the research questions, Chapter 4 and 7 are prototypical design chapters that aim at describing the framework and its implementation.

validation; finally (iv) pre-trained models across hundreds of projects that can be used for future studies and comparisons.[11]

C_4 Chapter 6 contributes to this thesis with an empirical study to compare metrics-based and unsupervised-learning-based detectors for Blob blueprint, namely a too large IaC script often called *Blob* or *God class* in traditional code smell detection. The study corroborates previous findings and empirically shows that metric-based detectors perform better in detecting Blob blueprints. In addition, it provides a comprehensive dataset of publicly available TOSCA blueprints, including source code measurements calculated upon these blueprints and manually validated observations related to the Blob blueprint smell.[12]

C_5 Chapter 7 embraces the previous contributions and present the final implementation of the Machine Learning framework for IaC defect prediction defined in Chapter 4: DEFUSE.[13]

[11]https://github.com/stefanodallapalma/TSE-2020-05-0217
[12]https://github.com/jade-lab/tosca-smells
[13]https://github.com/radon-h2020/radon-defuse

Note

This research is part of the European Union's Horizon research and innovation project called RADON (https://radon-h2020.eu/). The project aims to achieve a broader adoption of serverless computing technologies within the European software industry. To achieve this, the project provides a DevOps framework to create and manage microservice applications that can exploit serverless computing. Once a serverless application enters the RADON lifecycle model, it is organized as a continuous DevOps loop comprising six critical phases: Verification, Decomposition, *Defect Prediction*, Continuous Testing, Monitoring, and Continuous Integration/Continuous Deployment.
Defect prediction stands between application development and testing to help users improve the quality of the codebase. It visualizes code metrics, localizes defects, and detects code smells, and ultimately prioritizes files for the testing phase (Dalla Palma et al. 2023).
In short, one RADON's key pillars is quality assurance tools for Infrastructure-as-Code. Therefore, this research contributes to this pillar by developing DEFUSE, for which we received encouraging feedback from the industrial partners involved in the project.

Both researchers and practitioners benefit from the contributions mentioned above. For example, researchers can use the shared material as a baseline to better understand failure-prone infrastructure code and compare competing approaches for defect prediction. Furthermore, researchers gain valuable knowledge of novel IaC languages and tools, comprehensive datasets and metrics sets, and prediction models. In short, this research puts a clean baseline for existing and future approaches.

1.5. THESIS OUTLINE

Chapter 2 provides the necessary background information to contextualize the research and presents an overview of the related work on infrastructure code quality. The following five chapters delve into the specific contributions of this research, each addressing a distinct aspect of infrastructure code quality management.

- **Chapter 3** – *A Catalog of Software Quality Metrics for Infrastructure Code* introduces a catalog of quality metrics specifically designed for infrastructure code written in Ansible and showcases their usage to analyze infrastructure code. In developing these metrics, the work is often inspired by or transposes their definition from application code to infrastructure code. However, while quality metrics for traditional code have been widely investigated, these cannot necessarily be transferred, and respective new metrics have to be devised and evaluated.

- **Chapter 4** – *A Novel Framework for IaC Defect Prediction* proposes a fully integrated Machine Learning framework for predicting defects in repositories. The chapter describes the framework's various components, including repository crawling, metrics collection based on the metrics devised in Chapter 3, model building, and evaluation.

- **Chapter 5** – *A Large Empirical Study on Defect Prediction in Infrastructure Code* reports on an empirical study conducted to evaluate different classification techniques and features that the framework should use to achieve the highest detection accuracy in defect prediction. This chapter also shows how the infrastructure source code metrics proposed in Chapter 3 can be used and combined as surrogate metrics for defect-proneness of infrastructure components and to identify defects in Ansible.

- **Chapter 6** – *An Empirical Study on Blob Blueprint in TOSCA* reports on another empirical study conducted to evaluate which metrics-based or unsupervised learning techniques and features the framework should use to achieve the highest detection accuracy in smell detection.

- **Chapter 7** – *Defuse: A Data Annotator and Model Builder for Software Defect Prediction* provides a concrete implementation of the previously proposed framework and illustrates its application for defect prediction. The chapter showcases DEFUSE as a tool that utilizes the framework's concepts and provides examples of its usage. This separation between the conceptual framework in Chapter 4 and its implementation in this chapter allows for a better understanding of the contributions of the research, independent of any specific implementation.

Finally, Chapter 8 summarizes the lessons learned from the research, identifies open issues, and suggests future directions for quality management. Appendix A and Appendix B provide additional details on ANSIBLEMETRICS and REPOMINER, the two fundamental tools on which DEFUSE relies.

2
Background and Related Work

This chapter describes the background information to outline the context of this thesis and overviews the related work on infrastructure code quality. Section 2.1 overviews Ansible and Tosca by highlighting their syntax and characteristics. Section 2.2 introduces the terms and concepts related to software defect prediction required to understand the remaining of the thesis. Finally, Section 2.3 reviews previous works on Infrastructure-as-Code (IaC) quality, particularly in defect prediction and code smell detection.

2.1. DevOps and Infrastructure-as-Code

The DevOps methodology entails adopting organizational and technical practices, such as CI/CD, blending development and operation teams, to deal with the modern digital ecosystem and digital market that demands the "need for speed": fast and early releases, continuous software updates, constant evolution of market needs, and adoption of scalable technologies such as Cloud computing. In this context, IaC is the DevOps practice of describing and managing Cloud-based deployments using machine-readable configuration code.

The leading enabler for IaC has been the advent of Cloud computing, which has made the programmatic provisioning, configuration, and management of computational resources common practice. As a result, many languages and platforms have been developed, each dealing with specific aspects of infrastructure management: from tools able to provision and orchestrate virtual machines (Cloudify, Terraform), to those doing a similar job for container technologies (Docker Swarm, Kubernetes), to machine image management tools (Packer) and cloud orchestration (TOSCA), to configuration management tools (Chef, Ansible, Puppet).

Ansible has been gaining traction in recent years as an agent-less (i.e., no master node) and straightforward alternative to other more complex IaC technologies such as Chef and Puppet. At the same time, the "Topology and Orchestration Specification for

Listing 1 An example of Ansible playbook.

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

Cloud Applications" (TOSCA) builds upon existing configuration and orchestration languages to improve the readability and portability of configuration files across platforms. The following sections describe those two technologies.

2.1.1. ANSIBLE: AN OVERVIEW

Ansible is an automation engine based on the YAML language that automates cloud provisioning, configuration management, and application deployment, among others (Keating 2015). It works by executing scripts called modules, which describe or change the system state. While modules allow for the proper functioning of Ansible scripts, playbooks enable orchestrating multiple slices of the infrastructure topology, with detailed control of the architecture's scalability (e.g., how many machines to tackle at a time). Playbooks are essential for configuring and orchestrating deployment in Ansible. They can declare configurations and orchestrate any manual ordered process steps by launching tasks within one or more plays. Each play executes part of the overall goal of the playbook, running one or more tasks, which are calls to Ansible modules.

For example, Listing 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 is worth noting that a playbook runs in order from top to bottom. Within each play, tasks also run in order from top to bottom. Therefore, the playbook in the example first targets the web servers to ensure that the Apache server is at the latest version, then the database servers to ensure that PostgreSQL is at the latest version and started. It achieves this by mapping the hosts (lines 2 and 14) to their respective tasks

Listing 2 An example of a TOSCA blueprint.

```
tosca_definition_version: tosca_simple_yaml_1_0
description: Start Elasticsearch on a Virtual Machine

node_types:

  tosca.nodes.indigo.Elasticsearch:
    derived_from: tosca.nodes.SoftwareComponent
    properties:
      elastic_password:
        type: string
      kibana_password:
        type: string

  interfaces:
    Standard:
      configure:
        implementation: artifacts/elasticsearch_install.yml
        inputs:
          elastic_password:
            { get_property: [ SELF, elastic_password ] }
          kibana_system_password:
            { get_property: [ SELF, kibana_password ] }

topology_template:

  node_templates:

    elasticsearch:
      type: tosca.nodes.indigo.Elasticsearch
      properties:
        elastic_password: "Password for user elastic"
        kibana_password: "Password for kibana system user"
```

(lines 9-12, 19-22, 25-27). There, `yum` and `service` are modules to manage packages with the yum package manager and to control services on remote hosts, respectively; `name` – the name of the package and the database – and `state` – whether present, absent, or otherwise – are parameters of these modules. In short, composing a playbook of multiple plays enables orchestrating multi-machine deployments and run specific commands on the machines in the `webservers` and `databases` groups.

2.1.2. Tosca: An Overview

The Topology and Orchestration Specification for Cloud Applications (TOSCA) is "an OASIS open standard that defines the interoperable description of services and applications hosted on the cloud, including their components, relationships, dependencies, requirements, and capabilities, thereby enabling portability and automated management across cloud providers regardless of underlying platform or infrastructure."[1]

Unlike other configuration management tools, such as Puppet, Docker, and Ansible, TOSCA covers the complete application life-cycle, rather than just deployment and configuration management (Lipton et al. 2018).

The creator of a cloud service captures its structure in a service topology written in a YAML-based domain-specific language. The topology is a graph of nodes represent-

[1] `https://www.oasis-open.org/committees/tosca/faq.php` (Accessed July 2022)

Listing 3 The *elasticsearch_install.yml* in Listing 2 (line 17).

```yaml
- hosts: localhost
  connection: local
  vars:
    es_config:
      network.bind_host: "{{ bind_host|default('0.0.0.0') }}"
    es_api_username: elastic
    es_api_password: "{{ elastic_password|default('...') }}"
    es_users:
      native:
        kibana:
           password: "{{ kibana_password|default('...') }}"
  roles:
    - role: elastic.elasticsearch
```

ing the service components and relationships connecting and structuring nodes into the topology. Both nodes and relationships are typed and hold a set of type-specific properties. Types form reusable entities defining the semantics of the node or relationship, such as properties, attributes, requirements, capabilities, and interfaces. Templates form the cloud service's topology using these types; they define how the respective type is instantiated for use in the application, and define the start values of the properties by specifying their defaults. In addition, templates can overwrite and extend the types to adjust them for their respective application. These types are conceptually comparable to abstract classes in Java, while templates are comparable to concrete classes extending these abstract classes (Binz et al. 2014).

Listing 2 shows a blueprint to start Elasticsearch on a Virtual Machine.[2] The service structure is defined by a topology template (line 24) incorporating a node template (line 28) of type `tosca.nodes.indigo.Elasticsearch` defined at the top under `node_types` (line 6). Besides nodes and relationships, other essential components are capabilities, requirements, and interfaces. Nodes can offer specific capabilities to other nodes at run-time. For example, in Listing 2, an additional node can be added to offer the capability to host the `elasticsearch` node, which means that the node can function as a host for other nodes. On the contrary, nodes can have requirements describing the functionality required to be implemented. For example, in Listing 2, the `elasticsearch` node might require the additional node as a host.

Node also provides interfaces, i.e., sets of operations accompanied by a specific implementation on how to execute them. For example, the `elasticsearch` node in Listing 2 is configured by the `configure` interface (line 16); it specifies the script that performs this configuration (line 17) and its input parameters (lines 18-20). Although implementing this configuration is an Ansible playbook, other languages or technologies such as Python, Chef, or Puppet can be used.

[2]Adapted from `https://github.com/indigo-dc/tosca-types/blob/master/examples/elasticsearch.yaml` (accessed on Jan 2022)

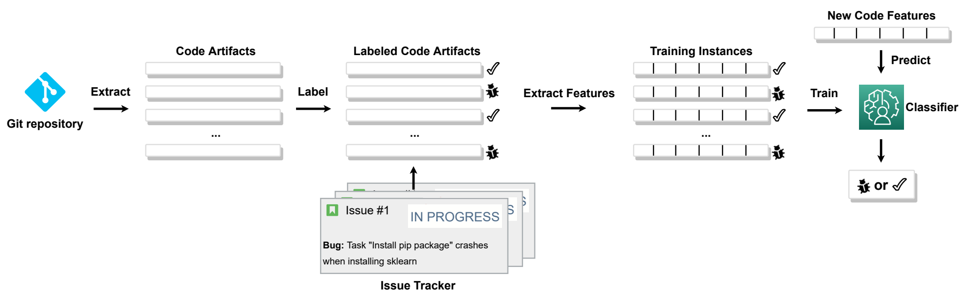


Figure 2.1: A typical defect prediction process. Code artifacts, e.g., blueprints, are extracted from software archives, such as git repositories. These artefacts are then labeled as failure-prone or neutral, or with the number of defects they contains, using information from commits and/or issue trackers connected to the software archive. Then, every instance is characterized using product and/or process metrics and used as training instances for a Machine Learning algorithm. Predicting a new instance consists of extracting the same metrics and pass them to the model for prediction.

2.2. Software Defect Prediction in a Nutshell

Software defect prediction supports software testing by identifying the parts of the system that are failure-prone and require extensive testing (Hall et al. 2012). The research literature presents several approaches to identifying defective code using supervised and unsupervised learning methods in within- and cross-project contexts (Paramshetti et al. 2014); a complete overview of state of the art is available in the systematic literature reviews conducted by Hall et al. 2012 and Hosseini et al. 2019.

In the *within*-project approach, a prediction model is trained using the project's historical data to predict new defects during the development of the same project, while in the *cross*-project approach, a model is trained on one project and applied to another.

These approaches can be further divided based on the prediction type at hand. For example, binary classification can be applied to check whether the code is failure-prone, while regression can be used to predict the number of defects present in the code.

DEFUSE defines binary classification models that classify IaC scripts as failure-prone or neutral in a *within*-project context. As such, the following concepts have to be introduced to understand the remaining of the thesis:

- **Defect** - An imperfection or deficiency causing the code not to meet its requirements or specifications (IEEE 2010).
- **Defect-fixing commit** - A commit that takes action related to a defect.
- **Defect-introducing commit** - A commit that contributes to introducing a defect.
- **Failure-prone script** - A script that presents defects and must be repaired or replaced, opposite of a *neutral* script.

As illustrated in Figure 2.1 and described in Zhiqiang et al. 2018, building Machine Learning-based defect predictors requires generating instances from software archives

Listing 4 A failure-prone Ansible script.

```
---
# File: main.yml - Main tasks for Consul

- name: Install python dependencies
  when:
    - consul_install_dependencies | bool
  block:
    - name: Install netaddr dependency on controlling host (with --user)
      pip:
        name: netaddr
        extra_args: --user
      delegate_to: 127.0.0.1
      become: false
      run_once: true
      when: is_virtualenv is not defined  # -- Wrong comparison
      # when: is_virtualenv == '' or is_virtualenv == None
```

such as version control systems, e.g., Subversion[3] or Git[4], and issue tracking systems, e.g., GitHub Issues[5], Bugzilla[6], and Jira[7]. The version control systems contain the source code and commit messages, while the issue tracking systems include defect information.

According to prediction granularity, those instances can represent a source code file, a class, a method, or a code change. Then, the defect information in the issue tracking system connected to the software archive is used to label an instance as failure-prone or neutral according to whether the instance contains defects, or its number.

Afterward, every instance is characterized using several defect prediction metrics (a.k.a., features). Defect prediction metrics are typically grouped into *product metrics* and *process metrics*. Product metrics are directly extracted from the source code and mainly measure aspects concerning the size and complexity of the source code. The ground assumption is that the source code with higher complexity can be more defect-prone. For example, the *lines of code* is a commonly used and representative size metric. Process metrics are extracted from the historical information archived in the version control and issue tracking systems. These metrics reflect the changes over time and quantify many aspects of the software development process, such as the number of code changes and developer information.

Finally, defect prediction models can be built using a set of training instances based on the obtained metrics and labels and trained to discriminate and predict the presence or the number of defects in a specific component.

2.3. Related Work in Infrastructure As Code Quality

Due to the software design and development paradigm shift, Infrastructure-as-Code has recently received increasing attention in the research community. However, as shown by previous research in IaC, like any other source code artifact, infrastructure configu-

[3]https://subversion.apache.org/
[4]https://git-scm.com/
[5]https://github.com/features/issues/
[6]https://www.bugzilla.org/
[7]https://www.atlassian.com/software/jira

ration management scripts can potentially be failure-prone. They may contain defects that preclude correct functioning (Rahman and Williams 2018; Rahman and Williams 2019). Indeed, errors might still occur at the infrastructure code level, for example, build failures, crash due to access to missing resources, wrong comparisons, and many more. For example, the comparison at line 15 in Listing 4 causes the system to fail. In this example, *"the playbook tries to run the task "Install netaddr dependency on controlling host (virtualenv)" when variable VIRTUAL_ENV is not set in VM. Then is_virtualenv is defaulted to an empty string what is treated as defined value. Then task without user arg is invoked, resulting in error caused by lack of permissions on VM."*[8] The fix requires checking the variable against an empty string or `None` (line 16).

Various works related to this research empirically investigated the adoption, challenges, and particularly defects of IaC. Rahman, Mahdavi-Hezaveh, et al. 2019 investigated the research challenges in IaC through a systematic literature mapping. They identified four principal research areas surrounding IaC: (i) frameworks and tools, (ii) use of IaC, (iii) empirical studies, and (iv) testing. They concluded that, besides those research areas, research in the context of IaC defects and security flaws is still in its early stages.

As for IaC defects and smells, only a few research studies were conducted on the current development practices of infrastructure code (Guerriero et al. 2019) or analyzed source code properties to evaluate the quality of infrastructure code (Sharma, Fragkoulis, et al. 2016; Bent et al. 2018; Rahman and Williams 2019). Jiang et al. 2015 analyzed the co-evolution between infrastructure and production code, finding that the former is tightly coupled with test files, leading testers to change infrastructure specifications often when modifying tests.

Most of the previous work describes code quality in terms of the smelliness and defects-proneness of software components described via Chef and Puppet configuration management technologies. This thesis builds on this line of research, and the studies described below, to define source code measures able to further model quality aspects of IaC and implement tools for infrastructure code quality focusing on defect prediction. The focus is on Ansible, rather than Puppet and Chef, for a two-fold reason: (i) Ansible is the most popular IaC language in the industry to date; (ii) at the best of our knowledge, no attempts to analyze source code properties for Ansible has been previously made.

2.3.1. Previous Work on Defect Prediction

As for defect prediction, Rahman, Partho, et al. 2018 investigated the challenges in developing IaC, specifically configuration management tools. First, they looked for the most-asked questions on Stack Overflow to help IaC developers. Also, in this case, the focus was on Puppet-related questions. By applying qualitative analysis, they identified the three most common question categories: syntax errors, provisioning instances, and assessing the Puppet's feasibility in accomplishing specific tasks. The three categories of questions that yielded the most unsatisfactory answers were installation, security, and data separation. The authors then classified IaC defects according to standard non-IaC defects categories by qualitatively analyzing commit messages and issue reports in open-source projects, limited to Puppet code.

[8] https://github.com/ansible-community/ansible-consul/issues/161

Category	Description
Conditional	Defects due to erroneous logic or conditional values used to execute one or multiple brancehs of an IaC script.
Configuration Data	Defects due to erroneous configuration data that reside in IaC scripts.
Dependency	Defects that occur when executing an IaC script depends upon an artifact that is either missing or incorrectly specified.
Documentation	Defects that occur when incorrect information about IaC scripts are specified in source code comments, in maintenance notes, and in documentation files such as README files.
Idempotency	Defects that violate the idempotency property for IaC scripts. Idempotency ensures that even after multiple execution the provisioned system environment is exactly the same as it was after the first execution of the relevant IaC scripts.
Security	Defects that violate confidentiality, integrity or availability for the provisioned system.
Service	Defects related to improper provisioning and inadequate availability of computing services, such as load balancing services and monitoring services.
Syntax	Defects related to syntax in IaC scripts.

Table 2.1: The defect taxonomy for IaC defined in Rahman, Farhana, Parnin, et al. 2020.

Later on, Rahman, Farhana, Parnin, et al. 2020 performed a qualitative analysis of defect-related commits collected from open-source software repositories of the Openstack organization and categorized defects in the eight classes described in Table 2.1: “conditionals”, “configuration data”, “dependencies”, “documentation”, “idempotency”, “security”, “service” and “syntax”.

Besides, the articles that relate the most to this research were recently proposed by Rahman and Williams 2018 and Rahman and Williams 2019. The former uses text mining techniques, such as bag-of-word, upon textual features extracted from Puppet scripts and reports promising results for both techniques on three different datasets: Mozilla, OpenStack, and Wikimedia Commons. The latter identified ten source code measures that significantly correlate with defective infrastructure code scripts, such as properties to execute bash or batch commands, to manage file permissions and SSH keys, and to execute external scripts.

This research can be seen as complementary to those mentioned above. First, rather than focusing on textual metrics, *this thesis deeply investigates the value of a broad set of structural and process metrics on the failure-proneness of infrastructure code scripts*; second, we focus on Ansible and not on Puppet.

2.3.2. Previous Work on Code Smell Detection

Code smells reveal another harm in software development. The focus on code smells is relevant to this thesis as they might lead to sever defects in a program – although not always the case (Olbrich et al. 2010). For example, it is commonly agreed that large and complex software artifacts are more prone to defects.

Looking at code smells, Sharma, Fragkoulis, et al. 2016; Schwarz et al. 2018; and Rahman, Parnin, et al. 2019 applied the well-known concept to IaC. Although many code smells were proposed and studied for traditional programs (Palomba, Bavota, Penta, et al. 2014; Palomba, Panichella, et al. 2016; Tufano et al. 2017; Fowler 2018), IaC-specific smells, and their detection techniques remain unripe. Sharma, Fragkoulis, et al. 2016 and Schwarz et al. 2018 were the first to apply the concept of code smell in the context of IaC. In particular, they proposed a catalog of 13 *implementation* and 11 *design* configuration smells applicable to Puppet, and later 17 code smells peculiar for Chef. Table 2.2 shows the list of smells and their description. The catalog was then benchmarked against 4621 Puppet repositories. Interestingly, design smells showed higher average co-occurrence than implementation smells: one wrong or non-optimal design decision introduces many quality issues in the future, thereby showing that the notion of code smells also applies to IaC.

From there, the research area of code smells in IaC evolved. Schwarz et al. 2018 built upon Sharma's work by selecting the ten most frequently occurring smells and converting them into detection rules for a static code analysis tool designed for the Chef language. Furthermore, they categorized the code smells for their dependence on a specific technology, which facilitates mapping these smells to other languages in future research.

Later, Rahman, Parnin, et al. 2019 identified seven security smells for IaC found from a qualitative analysis of Puppet scripts in open-source repositories and comprise (i) granting admin privileges by default, (ii) empty passwords, (iii) hard-coded secrets, (iv) invalid IP address binding, (v) suspicious comments (such as "TODO" or "FIXME"), (vi) use of HTTP without TLS, and (vii) use of weak cryptography algorithms. Furthermore, they showed that these smells could persist for an extended period.

This research can be seen as complementary to those mentioned above, as it studies the impact of both metric-based and unsupervised-learning-based code smell detection for infrastructure code. Indeed, previous work extensively used metric-based smell detection methods only. These methods take source code as the input, detect a set of source code metrics that capture the characteristics of a set of smells, such as *lines of code*, *coupling*, and *cohesion*, and detect smells by applying a suitable threshold (Sharma and Spinellis 2018; Aniche et al. 2018). Nevertheless, these approaches suffer from subjective interpretations and threshold dependency; determining a suitable threshold is challenging, while Machine Learning techniques were found promising to overcome shortcomings such as determining threshold values but need more investigation (Fontana et al. 2016; Di Nucci, Palomba, Tamburri, et al. 2018). However, because not yet been explored in IaC, it is unknown to what extent they can distinguish between *sound* and *smelly* blueprints. In addition, the number of conducted empirical validation studies is marginal, and, although recent work expands to other IaC languages such as Ansible, most empirical studies on IaC quality are limited to Chef and Puppet, which makes generalizability hard. Finally, to our knowledge, no publications related to configuration smells in TOSCA exist, despite the language being the IaC industry standard.

2.4. SUMMARY

This chapter described the syntax and features of the two IaC languages considered in this thesis: Ansible and TOSCA. In addition, it introduced background concepts and terms related to infrastructure code quality and the related literature in IaC defect prediction and code smell detection.

The following chapters build on the related literature to define infrastructure code quality metrics and frameworks that could enable IaC developers to maintain and evolve software-defined infrastructures during Quality Assurance effectively.

Code Smells	Description
Implementation Smells	
Missing Default Case	A default case is missing in a case or selector statement.
Inconsistent Naming Convention	The used naming convention deviates from the recommended naming convention.
Complex Expression	A program contains a difficult to understand complex expression.
Duplicate Entity	Duplicate hash keys or duplicate parameters present in the configuration code.
Misplaced Attribute	Attribute placement within a resource or a class has not followed a recommended order, e.g., mandatory attributes should be specified before the optional attributes.
Improper Alignment	The code is not properly aligned (such as all the arrows in a resource declaration) or tabulation characters are used.
Invalid Property Value	An invalid value of a property or attribute is used (such as a file mode specified using 3-digit octal value rather than 4-digit).
Incomplete Tasks	The code has “fixme” and “todo” tags indicating incomplete tasks.
Deprecated Statement Usage	The configuration code uses one of the deprecated statements such as “import”.
Improper Quote Usage	Single and double quotes are not used properly; for example, boolean values should not be quoted and variable names should not be used in single quoted strings.
Long Statement	The code contains long statements (that typically do not fit in a screen).
Incomplete Conditional	An “if..elsif” construct used without a terminating “else” clause.
Unguarded Variable	A variable is not enclosed in braces when being interpolated in a string.
Long Resource	Execute and bash Resources being too long.
Improper Quote Usage	Usage of variables in single quoted strings and to quote resource titles, with the exception of variables as resource titles.
Hyphens	The use of hyphens should be avoided.
Design Smells	
Multifaceted Abstraction	Each abstraction (e.g., a resource, class, “define”, or module) should be designed to specify the properties of a single piece of software.
Unnecessary Abstraction	When a class, “define”, or module does not contain declarations or statements specifying the properties of a desired system, such as empty classes, “define”, or modules.
Imperative Abstraction	An abstraction containing numerous imperative statements such as “exec”, which defies the purpose of the language.
Missing Abstraction	When resource and language elements are declared and used without encapsulating them in an abstraction such as a class or “define”, thus limiting reusability.
Insufficient Modularization	An abstraction suffers from this smell when it is large or complex and thus can be modularized further.
Duplicate Block	A duplicate block of statements more than a threshold indicates that probably a suitable abstraction definition is missing.
Broken Hierarchy	The use of inheritance must be limited to the same module. The smell occurs when the inheritance is used across namespaces without following a “is-a” relationship.
Unstructured Module	Each module in a configuration repository must have a well-defined and consistent module structure.
Dense Structure	This smell arises when a configuration code repository has excessive and dense dependencies without any particular structure.
Deficient Encapsulation	This smell arises when a node definition or External Node Classifier declares a set of global variables to be picked up by the included classes in the definition.
Weakened Modularity	Each module must strive for high cohesion and low coupling. This smell arises when a module exhibits high coupling and low cohesion.
Avoid Comments	Presence of comments in the code, except for license info.
Too many Attributes	The code use attributes excessively.
Law of Demeter	A Chef cookbook A includes another Chef cookbook B, but both of them include a cookbook C.
Include Consistency	The smell searches for transitive includes of cookbooks where the cookbook name suggests a similar functionality.
Empty Default	File default.rb should be empty and should not include any includes.

Table 2.2: Code smells in IaC presented in Sharma, Fragkoulis, et al. 2016 and Schwarz et al. 2018.

3 A Catalog of Software Quality Metrics for Infrastructure Code[1]

Various tools and languages characterize the Infrastructure-as-Code (IaC) landscape, often overlapping in terms of their goals. Hence, it is crucial to study their adoption to identify the de-facto standard ways to write IaC. In this context, defining infrastructure code quality metrics could enable IaC developers to maintain and evolve them during Quality Assurance effectively.

This chapter builds upon this line of research and defines a catalog of 46 source code measures for Ansible that model quality aspects of IaC and showcase how they can be used to analyze infrastructure code. The focus on Ansible is two-fold: (i) Ansible is the most popular IaC language on GitHub to date and one of the most used configuration management technology in the industry (Guerriero et al. 2019); (ii) no previous studies attempted to analyze source code properties for Ansible.

The advantages of a metrics-based quality management approach to infrastructure code are manifold, among others:

- The analysis of IaC properties can help developers understand and improve the quality of their infrastructure through incremental refactoring instead of the conventional trial-and-error approach.

- Source code properties can be used as early indicators of faulty infrastructure scripts, potentially leading to expensive infrastructure failures.[2]

[1]This chapter was published in: **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, 2020.*

[2]`https://www.cloudcomputing-news.net/news/2017/oct/30/glitch-economy-counting-cost-software-failures/`

- Specific metrics can be defined across IaC languages to understand the mutual and combined characteristics of infrastructure code blends instead of focusing on a single vendor-locked IaC solution.

This work is the first to elicit a broad set of quality measures that describe and quantify the characteristics of infrastructure code to support IaC developers in maintaining and evolving it. As such, it poses the basis for a more extensive evaluation of infrastructure code quality. It is worth noting that the catalog is broad in scope to allow for possible further studies on the relation between the proposed metrics and the quality of infrastructure code. The following sections describe the methodology used to elicit those measures and the catalog in detail.

3.1. METHODOLOGY

Software measurement is a quantified attribute of a characteristic of a software product (Fenton et al. 2014). One method to perform software measurement is to use a set of metrics to analyze the code itself, such as the number of lines and functions in a single file and the number of files in an application.

To elicit the metrics, we first looked for traditional and language-agnostic source code metrics potentially applicable to IaC, given its novelty. We stemmed from a survey of almost 300 traditional and object-oriented source code metrics (Nuñez-Varela et al. 2017), such as executable, commented, blank lines of code, function count, class entropy complexity, and average method size. Nevertheless, unfortunately, most of the object-oriented metrics do not apply to IaC: only eight are present in the proposed catalog. Then, we ported to Ansible some of the metrics applicable to IaC that Rahman and Williams 2019 introduced for Puppet. Finally, we consulted the official Ansible documentation[3] to search metrics specific to IaC scripts written in Ansible: from the *atomic* structural characteristics for which structural metrics were directly implementable to the more complex ones that spread through multiple scripts or can be expressed as a combination of atomic measures. Those cover most of the constructs related to playbooks and tasks, such as plays, tasks, and modules, and include metrics dealing with error handling, bad and best practices (e.g., using deprecated statements and naming tasks uniquely, respectively), access to data from outside sources, and more.

The first two authors of the study scanned each resource to elicit an initial set of metrics that were possibly suitable for measuring IaC code quality. In this step, they considered two selection criteria:

(i) The metric must be elicitable from the source analyzed, e.g., when reading the Ansible documentation. The first author ensured that the description was accurate enough to define a metric.

(ii) The metric must enable discussions on its potential impact on infrastructure code quality.

Then, they opened a joint discussion and enacted a card-sorting exercise (Lewis et al. 2010) with the remaining authors of the study to refine the catalog, establish the impact

[3]Ansible documentation [4]

Name	Measurement technique
LinesBlank	Total empty lines
LinesComment	Count statements starting with `#`
LinesSourceCode	Total lines of executable code
NumConditions	Count of `is`, `in`, `==`, `!=`, `>`, `>=`, `<`, `<=` occurrences in `when`
NumDecisions	Count of `and`, `or`, `not` syntax occurrences in `when`
TextEntropy	$-\sum_{t \in tokens(script)} p(t) log_2(p(t))$
NumTasks	len(`tasks`) in playbook or tasks file
AvgTaskSize	*LinesSourceCode*(tasks)/NumTasks

Table 3.1: Traditional metrics applicable to IaC.

that each metric may have on the quality of infrastructure code, and assign names or scopes to the metrics: if a metric came from existing papers which already named it, they kept the same name; otherwise, they assigned a new, self-explainable (whenever possible) name. Disagreements on metrics names or rationale were solved immediately. Afterward, the authors grouped metrics and discussed those to consider as language-agnostic. Again, disagreements were solved immediately.

At the end of the process, metrics were classified into three groups:

- Object-oriented metrics that apply to Ansible and IaC in general.
- Metrics investigated in previous works on IaC and applicable to Ansible.
- Metrics related to best and bad practices in Ansible reported in the documentation and external resources and books.

The metrics in the first two groups were investigated regarding their value to code quality, although some of them were not yet studied in the context of IaC; the latter group emerged when analyzing the recommendations to design quality infrastructure code.

3.2. A CATALOGUE OF METRICS FOR IAC

The catalog consists of 46 code metrics, corresponding to the three categories mentioned above:

- Eight metrics relate to code characteristics language-agnostic.
- Fourteen metrics have been adapted from those previously developed by Rahman and Williams 2019 for Puppet.
- Twenty-four metrics concern some inherent characteristics of Ansible observable in different orchestration configuration languages.

3.2.1. TRADITIONAL METRICS APPLICABLE TO IAC

The first group of metrics is listed below; Table 3.1 summarizes them and their measurement technique. They all concern the characterization of long or complex infrastructure

Listing 5 Example of an Ansible playbook where metrics related to language-agnostic code characteristics can be extracted.

```
1  - name: Shut down systems
2
3    tasks:
4    - name: Shut down CentOS 6 systems
5      ansible.builtin.command: /sbin/shutdown -t now
6      when:
7        - ansible_facts['distribution'] == "CentOS"
8        - ansible_facts['distribution_major_version'] == "6"
9
10   - name: Shut down Debian 7 systems
11     ansible.builtin.command: /sbin/shutdown -t now
12     when: ansible_facts['distribution'] == "Debian" and
13           ansible_facts['distribution_major_version'] == "7"
```

code. As widely reported in the literature on source code quality, those metrics could make the code more failure-prone (D'Ambros et al. 2012b; Zhang 2009). While no empirical evaluation of the impact of these metrics is still available in the context of IaC, we hypothesize that similar conclusions could be reached.

- LinesSourceCode, LinesComment, and LinesBlank – Count the total number of *executable lines of code*, *lines of comments*, and *blank lines*, respectively.
  Interpretation: the more lines of code, the more complex and challenging to maintain the blueprint.

- NumConditions and NumDecisions – *Conditions* are Boolean expressions containing no Boolean operators (e.g., `and` and `or`); *decisions* are Boolean expressions containing conditions and one or more Boolean operators. In Ansible, those are primarily specified in the `when` statement.
  Interpretation: the more the conditions and decisions, the more complex and challenging it is to maintain the blueprint.

- TextEntropy – Measures the complexity of a script based on its information content, analogous to the *Class Entropy Complexity* (Nuñez-Varela et al. 2017).
  Interpretation: the higher the text entropy, the more challenging it is to maintain the blueprint.

- NumTasks – Measures the number of functions in a script, analogous to the traditional *Number of Methods Call* (Nuñez-Varela et al. 2017). Ansible tasks are considered methods, as they execute a modules with specific arguments.
  Interpretation: the higher the number of tasks, the more complex and challenging to maintain the blueprint.

- AvgTaskSize - Measures the average size of program modules and is analogous to the traditional *Average Method Complexity* (Nuñez-Varela et al. 2017).
  Interpretation: the higher this measure, the more complex and challenging to maintain the blueprint.

For example, Listing 5 has 13 lines, eleven being executable code and two blank lines. Hence, LinesCode = 11, LinesBlank = 2. Then, NumTasks = 2 (lines 4 and 10); being

Name	Measurement technique
NumCommands	Count of `command`, `expect`, `psexec`, `raw`, `script`, `shell`, and `telnet` syntax occurrences
NumEnsure	Count of `\w+.stat.\w+ is defined` regex matches
NumFile	Count of `file` syntax occurrences
NumFileMode	Count of `mode` syntax occurrences
NumImportPlaybook	Count of `import_playbook` syntax occurrences
NumImportRole	Count of `import_role` syntax occurrences
NumImportTasks	Count of `import_tasks` syntax occurrences
NumInclude	Count of `include` syntax occurrences
NumIncludeRole	Count of `include_role` syntax occurrences
NumIncludeTasks	Count of `include_tasks` syntax occurrences
NumIncludeVars	Count of `include_vars` syntax occurrences
NumParameters	Count the keys of the dictionary representing a module
NumURLs	Count of `url` syntax occurrences
NumSSH	Count of `ssh_authorized_key` syntax occurrences

Table 3.2: Metrics investigated in different IaC languages and ported to Ansible.

the tasks grouped in a single play, AVGTASKSIZE = 2. In addition, NUMCONDITIONS = 4 and NUMDECISIONS = 2.

3.2.2. METRICS INVESTIGATED IN DIFFERENT IAC LANGUAGES

Follows the second group of metrics listed below; Table 3.2 summarizes them and their measurement technique. Those metrics were generalized from the previous work conducted by Rahman and Williams 2019, where the authors observed a significant correlation with defective infrastructure code scripts. Specifically, they conducted a qualitative analysis with practitioners and empirically validated such metrics in the scope of defect prediction of Puppet code. A common interpretation of the following metrics is that higher values increase defect-proneness.

- NUMCOMMANDS – Puppet allows executing external commands via the resource type `exec`. Ansible provides the modules `command`, `expect`, `psexec`, `raw`, `script`, `shell`, and `telnet` for the same functionality.
  Interpretation: the number of external commands makes the blueprint more complex and challenging to maintain.

- NUMENSURE – Puppet allows checking the existence of a file, directory, or symbolic link via the resource type `ensure`. For the same functionality, Ansible provides the module `stat`.
  Interpretation: blueprints with many file existence checks are less prone to misbehavior but more challenging to test.

- NUMFILE – Puppet allows managing files, directories, and symbolic links via the resource type `file`. For the same functionality, Ansible provides the module `file`.
  Interpretation: the higher this metric, the more challenging it is to maintain and test the blueprint.

- NUMFILEMODE – `mode` is a source code property used to set permissions of files. It exists either in Puppet (as an attribute of the `file` resource type) or Ansible (as a parameter of the `file` module).
  Interpretation: higher levels of this metric make the blueprint less prone to misbehavior but more challenging to test.

- NUMINCLUDE – In Puppet, other scripts can be executed with the `include` function. This functionality in Ansible is provided by several `include` and `import` modules that allow users to break up large playbooks into smaller files, which can be used across multiple playbooks or even multiple times within the same playbook. `Import` statements are pre-processed at compilation time and consist of:

  - NUMIMPORTPLAYBOOK – `import_playbook` is used to include a file with a list of plays to be executed in the current playbook.
  - NUMIMPORTROLE – `import_role` is used to load a role when the playbook is parsed.
  - NUMIMPORTTASKS – `import_tasks` is used to import a list of tasks to be added to the current playbook for subsequent execution.

  `Include` statements are processed at execution-time and consist of:[5]

  - NUMINCLUDE – `include` is dynamically used to load and execute a specified role as a task.
  - NUMINCLUDEROLE – `include_role` is used to dynamically load and execute a specified role as a task.
  - NUMINCLUDETASKS – `include_task` is used to include a file with a list of tasks to be executed in the current playbook.
  - NUMINCLUDEVARS – `include_vars` is used to load YAML or JSON variables from a file or directory recursively during task run-time.

  Interpretation: the more the includes and imports, the more reusable but the more challenging to maintain the blueprint.

- NUMPARAMETERS – In Puppet, the state of a resource is described with an attribute. Similarly, in Ansible, module *parameters* describe the desired system state.
  Interpretation: the more the parameters, the more challenging to debug and test the blueprint.

- NUMSSH – Puppet provides the source code property `ssh_authorized_key` to manage SSH authorized keys. Similarly, Ansible provides the module `authorized_key` to add or remove SSH authorized keys for particular user accounts.
  Interpretation: the more this property, the more challenging to maintain and test the blueprint.

[5]https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_includes.html

Listing 6 Some of the source code properties designed by Rahman and Williams 2019 for Puppet and ported to Ansible.

```
1  - name: Perform some preliminar tasks
2    import_tasks: stuff.yaml
3
4  - name: Retrieving file status
5    stat:
6      path: /etc/foo.conf
7    register: conf
8
9  - name: Change file permission if exists
10   file:
11     path: /etc/foo.conf
12     mode: '0644'
13   when: conf.stat.exists is defined and conf.stat.exists
```

Name	Measurement technique
NumBlocks	Count of `block` syntax occurrences
NumBlocksErrorHandling	Count of `block-rescue-always` section occurrences
NumDeprecatedKeywords	Count the occurrences of deprecated keywords
NumDeprecatedModules	Count the occurrences of deprecated modules
NumDistinctModules	Count of *distinct* modules maintained by the community
NumExternalModules	Count the modules not maintained by the community
NumFactModules	Count occurrences of fact modules (listed in the doc)
NumFilters	Count of | syntax occurrences inside {{*}} expressions
NumIgnoreErrors	Count of `ignore_errors` syntax occurrences
NumLookups	Count of `lookup(*)` occurrences
NumNameWithVariables	Count of `name` occurrences matching the `".*{{\w+}}.*"` regex
NumUniqueNames	Count of `name` syntax occurrences with unique values

Table 3.3: Bad and Best IaC practice-related metrics.

- NUMURLS – URLs specify configurations. Ansible provides a module called `uri` to interact with HTTP(S) web services and requires setting a parameter `url`. *Interpretation:* the more the URLs, the more challenging to maintain and test the blueprint.

For example, Listing 6 shows a code snippet on which the metrics above are computed. In that code snippet, NUMINCLUDE = 1 (line 2), NUMPARAMETERS = 3 (lines 6, 11, 12), NUMFILE = 1 (line 10), NUMFILEMODE = 1 (line 12), and NUMENSURE = 1 (line 13). Also, NUMCONDITIONS = 2 and NUMDECISONS = 1, given the two conditions bound by logic *and* at line 11.

3.2.3. BEST AND BAD PRACTICE-RELATED IAC METRICS

The third group of metrics is listed in Table 3.3 and described below. It derives from the Ansible documentation and relates to best and bad practices and (external) data management. In general, these metrics could affect the comprehensibility and maintainability of infrastructure code.

- DEPRECATEDKEYWORDS and DEPRECATEDMODULES – Using deprecated modules and keywords is discouraged as they are kept for backward compatibility only.
  Interpretation: the more deprecated keywords and modules, the more difficult it is to maintain and evolve the code and the higher the likelihood of crashing if the system does not support retro-compatibility.

- NUMBLOCKS and NUMBLOCKSERRORHANDLING – Blocks group tasks logically and allow exception handling by appending a `rescue` clause or an `always` clause to each block. Tasks in *blocks* eventually execute; tasks in *rescue* sections execute when errors raise; tasks in *always* sections execute every time (i.e., including in case of errors).
  Interpretation: the more blocks, the easier code maintenance. Nevertheless, many blocks without `rescue` or `always` clause increase the chance of system crashes when incurring in wrong behaviors.

- NUMDISTINCTMODULES – Modules are reusable and standalone scripts called by tasks. They enable changing or getting information about the system state and can be interpreted as a degree of responsibility of the blueprint. Hence, a blueprint with many distinct modules is less self-contained and potentially affects system complexity and maintainability because it is responsible for executing many different tasks. Conversely, one could expect a task to execute several times with different options, for example, to ensure the presence of dependencies in the system.
  Interpretation: the more distinct modules, the more challenging it is to maintain the blueprint.

- NUMEXTERNALMODULES – Many modules are maintained by the community. However, users can create and maintain their modules, called *external modules*. Although modules maintained by the community must be fully documented and tested, this is not needed for external modules. Therefore, a blueprint with many external modules may be more challenging to maintain than a blueprint containing only modules maintained by the community.
  Interpretation: the more external modules, the more challenging it is to maintain the blueprint and the higher the chance of the system misfunctioning.

- NUMFACTMODULES – Fact modules do not alter the system state but only return data. For that reason, blueprints with many fact modules may be less prone to unexpected behaviors and easier to test.
  Interpretation: the more `fact` modules, the easier to test and maintain the blueprint.

- NUMFILTERS – Filters transform the data of a template expression, for example, to format or render them. Although they allow transforming data in a very compact way, filters can be concatenated to perform a sequence of different actions, as shown in Listing 7. This aspect may potentially affect the readability and maintainability of the code.
  Interpretation: the more filters, the lower the readability and the more challenging it is to maintain the blueprint.

Listing 7 An example of filters in Ansible. Note, the product filter returns the cartesian product of the input iterables, that is roughly equivalent to nested for-loops.

```
- name: generate multiple hostnames
  debug:
    msg: "{{['foo','bar']|product(['com'])|map('join','.')|join(',')}}"

# Output: { "msg": "foo.com,bar.com" }
```

- NUMIGNOREERRORS – Ansible provides different ways to handle errors. It is possible to prevent a playbook from stopping when a task fails by setting `ignore_errors: True`. However, ignoring errors is a bad practice because it hides error handling.
  Interpretation: the more the `ignore_errors:True` statements, the more the system is hard to debug and prone to misbehavior.

- NUMLOOKUPS – Lookups allow access to outside data sources and require an advanced working knowledge of Ansible plays before incorporating them. Besides, some lookups may pass arguments to a shell, and one should use them carefully to ensure safe usage.
  Interpretation: the more the lookups, the higher the chance of unexpected behavior and the more challenging it is to maintain the blueprint.

- NUMSUSPICIOUSCOMMENTS – Suspicious comments, such as TODOs, warn of defects, missing functionality, or system weaknesses.
  Interpretation: the more suspicious comments, the higher the chance of unexpected behavior because of missing or incomplete functionalities.

- NUMUNIQUENAMES – Uniquely naming plays and tasks is a best practice to quickly locate problematic plays and tasks. Duplicate names may lead to not deterministic or at least unclear behavior (Keating 2015).
  Interpretation: the more entities with unique names, the more maintainable and readable the blueprint.

- NUMNAMESWITHVARIABLES – With uniqueness as a goal, many playbook developers prefer to use variables instead of hard-coding names. This strategy may work well, but authors need to consider the source of the variable data they are referencing. Indeed, variable data can come from various locations, and the values assigned to variables can be defined many times. For the sake of play and task names, only variables for which the values can be determined at playbook parse time will parse and render correctly. If the data of a referenced variable is discovered via a task, the variable string will be displayed unparsed in the output (Keating 2015), potentially affecting debugging and software auditing.
  Interpretation: although names with variables could make more succinct code, they could hinder code debugging and potentially lead to unexpected behavior.

- NUMUSERINTERACTIONS – In some cases, an Ansible script requires the user input such as username and password to access a service. Asking for external input may affect the program's correctness at run-time. User interactions have to be handled

Name	Measurement technique
AvgPlaySize	*LinesSourceCode*(playbook) / *NumPlays*
NumKeys	Count of keys in the dictionary representing a playbook or tasks
NumLoops	Count of `loop` and `with_*` syntax occurrences
NumMathOperations	Count of `+`, `-`, `/`, `//`, `%`, `*`, `**` syntax occurrences
NumPaths	Count of `paths`, `src` and `dest` syntax occurrences
NumPlays	Count of `hosts` syntax occurrences
NumRegex	Count of `regexp` syntax occurrences
NumRoles	Length of the `roles` section in a playbook
NumSuspiciousComments	Count comments with `TODO`, `FIXME`, `HACK`, `XXX`, `CHECKME`, `DOCME`, `TESTME`, or `PENDING`
NumTokens	Count the words separated by a blank space
NumUserInteractions	Count of `prompt` syntax occurrences
NumVariables	Sum len(`vars`) in plays

Table 3.4: Size related measures. The higher the number of the these source code properties, the more complex the blueprint.

by the program with several conditions. If not handled properly, a given input may lead the system to crash at run-time.
Interpretation: the more user interactions, the higher the chance of unexpected behavior.

The remaining metrics (Table 3.4) are self-describing and measure different aspects of the size of a blueprint, which may affect its quality in terms of complexity and readability: NUMPLAYS, AVGPLAYSIZE, NUMROLES, NUMVARIABLES, NUMLOOPS, NUMMATHOPERATIONS, NUMPATHS, NUMREGEX (i.e., regular expressions), NUMTOKENS (i.e., words separated by blank spaces), and NUMKEYS (i.e., keys of the dictionary representing a playbook or a list of tasks). In particular, paths and regular expressions are often subject to typos, which might lead to run-time errors if they are not correctly handled. Generally, the higher the value of these source code properties, the more complex the blueprint.

Listing 8 shows an Ansible code snippet consisting of a list of tasks adapted from the Ansible documentation.[6] The tasks are grouped in two blocks: the first block (lines 3-6) is used to handle errors along with the `rescue` and `always` sections (lines 7 and 10); the second block (lines 15-26) is used to allow task execution only if the current system distribution is 'CentosOS' (line 26). Therefore, NUMBLOCKS = 2 and NUMBLOCKSERRORHANDLING = 1. The `ignore_errors:True` at line 20 prevents the playbook from stopping if the task fails to install the Apache server. The error should not be ignored but caught and handled in a `rescue` section. The word `TODO` in the comment is suspicious and suggests a possible issue. Hence, NUMIGNOREERRORS = 1 and NUMSUSPICIOUSCOMMENTS = 1. Other information that can be extracted relates to the number of distinct modules, namely `debug`, `command`, `yum`, and `service` (lines 3, 6, 17, 23) and the number of unique names. Note that the `name` on lines 18 and 24 are not task names but module parameters. Hence, NUMDISTINCTMODULES = 4 and NUMUNIQUENAMES = 5.

[6] https://docs.ansible.com/ansible/latest/user_guide/playbooks_blocks.html (Accessed on April 2020)

Listing 8 An example of properties inherent to Ansible.

```
- name: Attempt and graceful roll back demo
  block:
    - debug:
        msg: 'I execute normally'
    - name: I force a failure
      command: /bin/false
  rescue:
    - debug:
        msg: 'I caught the failure'
  always:
    - debug:
        msg: 'I always execute :-)'

- name: Install and start Apache
  block:
    - name:
      yum:
        name: httpd
        state: latest
      ignore_errors: True # TODO: handle error in rescue

    - name:
      service:
        name: bar
        state: started
  when: ansible_facts['distribution'] == 'CentOS'
```

3.3. SUMMARY

This chapter proposed a broad catalog of 46 code measures to evaluate the different aspects of IaC, the most comprehensive measures set for IaC to the best of our knowledge. The metrics can be categorized in scope (i.e., *general*, *playbook*, and *tasks list*) depending on the particular construct and artifact they target. For example, *general* metrics apply to either playbooks and files containing a flat list of tasks but can also generalize to other languages; *playbook* metrics operate within a single playbook, while *tasks list* metrics operate within a playbook and tasks files.

Although they target Ansible, many of the measures above are equivalent and portable to other languages (e.g., Chef, Puppet) and offer a general-purpose metrics-based approach for IaC quality evaluation. Currently, there is a lack of empirical evidence regarding the scope and usability of such measures, e.g., whether they can be used and combined to detect code smells and bugs. Besides, the definition of the proposed catalog would allow a fair comparison between traditional process metrics and IaC-oriented product metrics in the scope of defect prediction of infrastructure code. In this context, Chapter 5 shows how the proposed infrastructure source code properties can be used and combined as surrogate metrics for defect-proneness of infrastructure components and to identify defects in Ansible.

4

A Novel Framework for Infrastructure-as-Code Defect Prediction[1]

As shown by previous research and mentioned in previous chapters, like any other source code artifact, infrastructure configuration management scripts can be prone to failures (Rahman and Williams 2018; Rahman and Williams 2019; Rahman, Farhana, and Williams 2020). Therefore, the effective prediction of failure-prone scripts may enable organizations embracing the DevOps methodology to focus on such critical scripts during Quality Assurance and allocate effort and resources more efficiently. To this end, this chapter proposes a fully integrated Machine Learning-based framework for Infrastructure-as-Code (IaC) Defect Prediction that allows for repository crawling, metrics collection, model building, and evaluation.

Figure 4.1 provides a detailed overview of the proposed framework consisting of four individual components:

- The IaC Repositories Collector collects active IaC repositories on GitHub.
- The Repository Scorer computes repository metrics based on best engineering practices used to select relevant repositories.
- The IaC Repository Miner mines failure-prone and neutral IaC scripts from a repository. Then, it gathers a broad set of metrics from the literature comprising traditional application code metrics (e.g., lines of code), IaC-oriented metrics (e.g., number of configuration tasks), and process metrics (e.g., number of commits to a file). Finally, those metrics are computed upon the collected IaC scripts to predict their failure-proneness.

[1]This chapter was published in: **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. *Transactions on Software Engineering, 2021.*

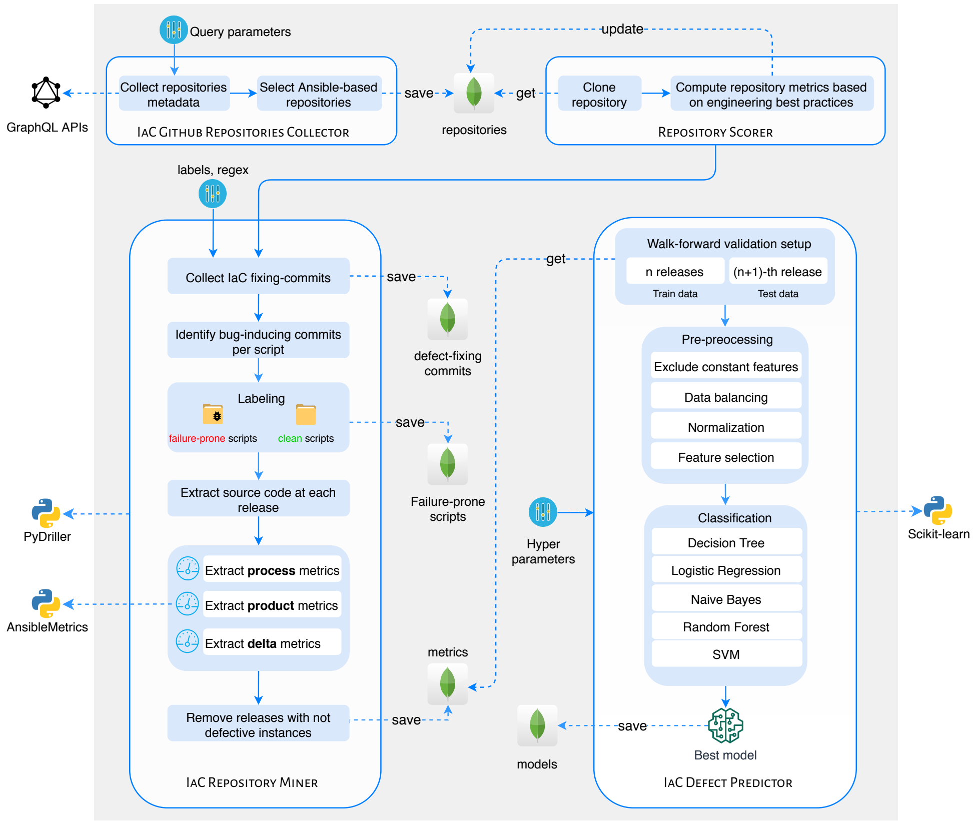


Figure 4.1: The proposed framework for IaC Defect Prediction. The repositories collected by the IaC Repositories Collector are passed as input to the Repository Scorer to pick relevant repositories. Afterward, the IaC Repository Miner mines the selected repositories; the generated dataset, consisting of observations of *failure-prone* and *neutral* IaC scripts for the individual repositories, is used by the IaC Defect Predictor to build and evaluate the models.

- The IaC Defect Predictor pre-processes the datasets and trains the Machine Learning models. Given an unseen IaC script, this component classifies it as *failure-prone* or *neutral.*

Contribution. The contribution of this chapter is the Machine Learning-based Framework for IaC defect prediction. A first implementation is released open-source on Github (`https://github.com/radon-h2020/radon-defuse`) and is further described in Chapter 7.

4.1. IaC Repositories Collector

Prior work on defect prediction for traditional application code uses datasets extracted from public software data archives (Fu, Menzies, and Shen 2016; Tantithamthavorn et

Table 4.1: Criteria to select repositories that contain evidence of engineered software projects.

	Criterion	Description
1	Push Events	The repository must have at least one *push event* to its default branch in the last six months, as evidence of recent development activity
2	Releases	The repository must have at least 2 releases since the proposed defect predictor analyzes files at each release and between successive releases
3	Ratio of IaC Scripts	At least 10% of the files must be IaC scripts to ensure that there are a sufficient number of IaC scripts to analyze
4	Core Contributors	The project must have at least 2 contributors whose total number of commits accounts for 80% or more of the total contributions, as evidence of collaboration
5	Continuous Integration	The repository must use a CI service, determined by the presence of a configuration file required by that service (e.g., a *.travis.yml* for TravisCI), as evidence of quality
6	Comment Ratio	The *comment ratio* must be at least 0.1%, as evidence of maintainability
7	Commit Frequency	The average number of commits per month must be at least 2.0, as evidence of sustained evolution
8	Issue Frequency	The average number of issue events transpired per month must be at least 0.01, as evidence of project management
9	License Availability	The repository must have evidence of a *license*, as evidence of accountability evolution
10	Lines of Code	The repository must have at least 100 *lines of code*; this criterion co-assesses and controls the criteria 4 to 9

al. 2016). However, only a handful of IaC datasets are publicly available and are limited to Chef and Puppet scripts. Therefore, the IaC REPOSITORIES COLLECTOR was implemented to search for *public* candidate repositories containing Ansible code through the novel GraphQL-based GitHub APIs.[2]

The tool is open-source and available on Github[3] and the Python Package Index[4] (PyPI). It enables tuning the GraphQL search query to collect metadata, such as name, description, URL, and root directories, from repositories that match user-defined selection criteria, such as the minimum number of releases, issues, stars, and watchers. In addition, archived, mirrored, and forked repositories are excluded. The metadata is used to select Ansible repositories by checking the word `ansible` against their *name* (e.g., `ansible/ansible-examples`), *description* (e.g., *"A few starter examples of ansible playbooks"*)[5], or *directory layout*, i.e., the presence of at least two of the following directories: `playbooks`, `meta`, `tasks`, `handlers`, `roles`. These metadata are then saved in a database and analyzed to mine only relevant projects.

4.2. REPOSITORY SCORER

A large portion of repositories on Github is not for software development but are mainly used for experimentation, storage, and academic projects (Kalliamvakou et al. 2014). Therefore, the framework verifies several constraints to mine only relevant projects; the constraints are reported in Table 4.1 along with their description and rationale.

Criterion 1 allows the crawler to discard inactive projects, while criterion 2 is needed

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

Algorithm 1 Procedure to identify defect-fixing commits.

```
procedure GETFIXINGCOMMITS
    DEFECTS = ['conditional', 'configuration data', 'dependency', 'docs', 'idempotency', 'security', 'service', 'syntax']

    fixingCommits = Dict[Str, List]

    for commit in $Repo.commits$ do
        fcc = FixingCommitClassifier(commit)

        for defect in DEFECTS do
            if fcc.fixes(defect) then
                fixingCommits[commit.hash].append(defect)

    discard_undesired_commits(fixing_commits)
    return fixing_commits
```

because the target models are trained at the release level. These criteria are evaluated by the IaC Repositories Collector, while the remaining criteria are evaluated by the Repository Scorer, a Python package implemented for the task and released open-source on Github[6] and PyPI[7]. More specifically, the *ratio of IaC Scripts* represents a cut-off to analyze repositories containing IaC scripts that have been determined by the previous works (Rahman and Williams 2018; Rahman and Williams 2019). Criteria 4-10 are considered good indicators of well-engineered software projects, i.e., *"software projects leveraging sound software engineering practices in one or more dimensions such as documentation, testing, and project management"* (Munaiah et al. 2017).

The tool requires the link to a Git repository, calculates metrics related to the last eight criteria reported in Table 4.1, and saves them in the database along with repository metadata. Once the repository is deemed relevant for the analysis, based on the computed metrics, its history is analyzed using the IaC Repository Miner.

4.3. IaC Repository Miner

IaC Repository Miner is a modular framework that combines failure data acquisition and metrics extraction to ease mining software repositories and create datasets for studies on defect prediction (see Appendix B). It builds on top of the well-known Python framework *PyDriller* (Spadini et al. 2018) to analyze the history of the projects in order to:

- Identify defect-fixing commits, i.e., commits that take action to remove a defect.
- Identify failure-prone scripts across the project's history and label them "failure-prone" or "neutral".
- Calculate process metrics and IaC-specific product metrics from those scripts and generate a dataset of observations to train defect prediction models.

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

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

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

4.3.1. Data Acquisition

To acquire failure data for training, IaC Repository Miner applies Algorithm 1 to identify *defect-fixing* commits (lines 7-12). Defects are categorized based on the qualitative analysis performed by Rahman, Farhana, Parnin, et al. 2020 on defect-related commits collected from open-source software repositories of the Openstack organization: “conditionals”, “configuration data”, “dependencies”, “documentation”, “idempotency”, “security”, “service” and “syntax”. Therefore, given a commit and a defect category, the function identifies whether the commit message or modifications to one of its files indicate a fix, based on the rules defined in (Rahman, Farhana, Parnin, et al. 2020). If so, the commit hash is added to the dictionary of defect-fixing commits with the respective categories (lines 11-12).

IaC Repository Miner keeps only the commits that modify at least one IaC file for the language (line 14) and returns the dictionary of defect-fixing commits.

Afterward, it determines their failure-proneness as follows. To this end, it applies Algorithm 2 to identify relevant files that are modified in the defect-fixing commits and their *bug-introducing* (or defect-introducing commit); commits are analyzed backward from the most recent defect-fixing commit to the oldest defect-fixing commit (line 4). Therefore, given a file, the algorithm ensures that it is written in the language selected for the analysis and has been modified; it is not a new, deleted, or renamed file (line 6).

Then, the *SZZ* algorithm[8] (Kim et al. 2006) is used to identify the **oldest** commit

[8]As implemented in *PyDriller* 2.0.

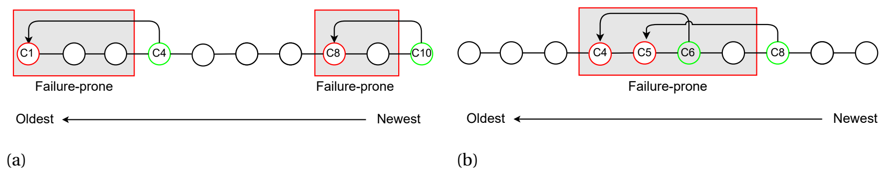


Figure 4.2: Two scenarios of the labeling process.

that introduced the defect in that file (line 9), known as the *bug-introducing* commit (*bic)*. The *bic*, the defect-fixing commit (*fic*), and the *filename* are used to create a new `FixedFile` object (line 11). The file is added directly to the list of fixed files (lines 15-16) when encountered for the first time through the algorithm. Successively,

- **Lines 19-20.** If the current defect-fixing commit (e.g., C4 in Figure 4.2a) is older than the file's previous *bic* (e.g., C8 in Figure 4.2a), then a brand new object (i.e., *FixedFile(file=A, fic=C4, bic=C1)*) is appended to the list of fixed files.

- **Lines 21-22.** If the file's previous *bic* (e.g., C5 in Figure 4.2b) is between its current *bic* (e.g., C4 in Figure 4.2b) and *fic* (e.g., C6 in Figure 4.2b), the existing *bic* is updated with the current one (i.e., from C5 to C4).

For the sake of clearness, lines handling file-renaming are omitted. Finally, Algorithm 2 returns the list of fixed files that are successively used to collect and label the files as failure-prone at a given point in the repository history. More precisely, all the snapshots of a fixed file between its *bic* (inclusive) and *fic* are labeled *failure-prone*.

Data Validation

To measure the soundness of the IaC Repository Miner in identifying *defect-fixing commits* of Ansible files, the authors of this work uniformly selected and manually assessed a statistically-relevant sub-sample of the commits identified by Algorithm 1 on the data collected in Section 5.1. Therefore, we applied a sampling technique to determine the appropriate sample size of fixing-commits that meet our confidence level and confidence interval (Kotrlik et al. 2001). First, the population of Ansible defect-fixing commits was set to the number of data points in the dataset obtained in Section 5.1 (i.e., 4,937). For this research, an acceptable confidence level was set to the generally accepted value of 0.05. As a confidence interval (*a.k.a.*, the acceptable margin of error), the generally accepted 5% was chosen. By applying Cochran's formula (Cochran 1977), we obtained a sample size of 357 commits. Cochran's formula requires a parameter representing the population's estimated proportion with the attribute in question. Because we do not know the actual distribution of defect-fixing commits, we used the worst-case scenario percentage of 50%, which yields the largest sample size. Next, we manually assessed the miner precision on the resulting 357 commits. The first author did the validation, while the second validated 15% of the samples. Finally, Cohen's Kappa (Cohen 1960) was measured to compute the degree of agreement between the assessors. In disagreements, the two assessors met and discussed the disagreed commits to convey a solution, i.e., label it either as a true positive or a false positive. If no agreement was reached, the source was sent to a third arbiter for further evaluation without stating the two previous assessors' decisions, thus avoiding convenience bias. The resulting third-arbiter decision emerges as the final one, following a majority vote.

Following the procedure above, we obtained a precision of 74%. We reached a complete agreement in the resolution phase from an initial Cohen's Kappa of 0.34 (79% agreement). We applied the same procedure to evaluate the precision of Algorithm 2 in identifying defect-introducing commits. The population of defect-introducing commits was set to the number of data points in the dataset obtained in Section 5.1 (i.e., 4,434). Then, we validated 354 defect-introducing commits, obtaining a *precision* of 84% and reaching complete agreement in the resolution phase from an initial Cohen's Kappa of 0.24 (79% agreement).

4.3.2. Metrics Extraction

IaC Repository Miner also gathers a comprehensive set of 108 features that can be used for training the defect prediction models.[9] Such features have been extracted from previous work in defect prediction (Rahman and Devanbu 2013; Rahman and Williams 2019; Dalla Palma, Di Nucci, Palomba, et al. 2020) and can be classified into three categories:

- **Infrastructure-as-Code Oriented (ICO) metrics** are structural properties derived from analyzing the IaC source code. From now on, we will use the terms ICO metrics and *product* metrics interchangeably. IaC Repository Miner *uses* the AnsibleMetrics tool (Dalla Palma, Di Nucci, and Tamburri 2020), described in detail in Appendix A, to collect the 46 metrics belonging to the catalog of IaC quality metrics, introduced in Chapter 3, that measure and potentially predict the maintainability of IaC scripts. More particularly, 8 of them are "traditional" code metrics that can be adapted for IaC scripts, e.g., the lines of code: these metrics have been long adopted in the context of traditional defect prediction and, therefore, we selected them to assess their role for predicting IaC defects. Other 14 metrics in this set have already been studied by Rahman and Williams 2019 for predicting defects in Puppet code: as such, we considered them to verify their generalizability when employed for predicting Ansible defects. Finally, the last 24 refer to best and bad practices and data management of Ansible code: in Chapter 3, we conjectured that these metrics could negatively affect the maintainability of IaC code and increase its failure-proneness. For example, let us consider the number of distinct modules: IaC scripts consisting of many distinct modules are naturally less self-contained and potentially affect the complexity and maintainability of the system; therefore, we considered it among the set of metrics for IaC defect prediction.

- **Delta metrics** capture the amount of change in a file between successive releases. Delta metrics have been associated with the failure-proneness of traditional source code by Arisholm et al. 2010. We selected them to assess their usefulness when employed in the context of IaC scripts and collected one delta metric for each IaC-oriented metric.

[9]A description of those metrics is reported at `https://github.com/stefanodallapalma/TSE-2020-05-0217/blob/master/METRICS.md`

Step	Options
Feature selection	Constant variables removal, Recursive Feature Elimination
Data balancing	None, Random under sampling, Random over sampling
Data normalization	None, Min-max normalization, Standardization
Classification	Decision Tree, Logistic Regression, Naive Bayes, Random Forest, Support Vector Machine

Table 4.2: Configurations used to build the defect prediction models.

- **Process metrics** consider aspects concerning the development process rather than the code itself. IaC Repository Miner *extends* PyDriller to collect 16 process metrics such as the number of developers that changed a file, the total number of added and removed lines, and the number of files committed together.[10] Also in this case, the selected metrics have been previously associated with the failure-proneness of traditional code (Moser et al. 2008; Rahman and Devanbu 2013) and was our willingness to experiment with them in a different context.

These metrics are extracted from IaC scripts at each release and saved in the database for analysis or use by the IaC Defect Predictor.

4.4. IaC Defect Predictor

The IaC Defect Predictor relies on the Python frameworks *scikit-learn* (Pedregosa et al. 2011) and *imblearn* (Lemaître et al. 2017) to build the pipeline that balances and pre-processes the dataset, trains, and validates the Machine Learning models, and uses it to predict unseen instances. In particular, it uses different configurations of feature selection, normalization, data balancing, classifiers, and hyper-parameters, as described below and in Table 4.2.[11]

1. *Feature selection.* The data have been intended for defect prediction in our study. Nevertheless, not all the dataset features may help the task because they are constant or do not provide valuable information exploitable by a learning method for a particular dataset. Therefore, the proposed framework uses feature selection to reduce the dataset's size, speed up the training, and select the optimal number of features that maximize a given performance criterion.

2. *Data balancing.* Class imbalance is a significant obstacle to proper classification by supervised learning algorithms (Batista et al. 2004). It is particularly true in defect prediction, where the *neutral* class outnumbers the *failure-prone* class. Balancing the training dataset is a well-known practice for supervised learning problems to overcome this obstacle. Therefore, once feature selection is finished, the training data are balanced such that the number of failure-prone instances equals the number of neutral instances. The proposed framework uses three configurations for balancing: (i) no balancing, (ii) random under-sampling of the majority

[10] We used the implementation available online at release 1.13. The process metrics are documented at `https://pydriller.readthedocs.io/en/latest/processmetrics.html`

[11] Given the number of classifiers and hyper-parameters, we preferred reporting the latter in the online appendix.

class, and (iii) random over-sampling of the minority class. The *imblearn* package implements both techniques (ii) and (iii) in Python.[12] An attempt was made to balance data with different techniques such as SMOTE, ADASYN, and Tomek-Links, but in most cases, their impact on the prediction was small compared to the increase in computational time. We, therefore, resorted to the faster random-sampling techniques.

3. *Data normalization.* In this step, the training data are normalized, scaling numeric attributes. The framework uses three configurations for data normalization: (i) no normalization, (ii) *min-max* transformation to scale each feature individually in the range [0, 1], and (iii) *standardization* of the features by removing the mean and scaling to unit variance.

4. *Classification.* The normalized data and the learning algorithm are used to build the learner. Before the learner is tested, the original test data are normalized, and the dimensionality is reduced to the same subset of attributes from step 1. After comparing the prediction with the actual value of the test data, the performance of one validation *pass* is obtained. Note that, in our framework, the classification step can be applied with any machine learning algorithm, i.e., the learner selection is left to the user.

The final output consists of a CSV file that reports the performance of the models for each validation step and a *.joblib* file for each trained model to persist them for future use without retraining.

4.5. SUMMARY

This chapter presented a fully integrated Machine Learning framework for IaC defect prediction that allows for repository crawling, metrics collection, model building, and evaluation. A first implementation is available on GitHub.[13] In the next chapter, we evaluate the framework in a within-project setup using supervised Machine Learning techniques to detect defects in IaC scripts early and learn features that characterize them from the individual projects considered.

[12] `https://imbalanced-learn.readthedocs.io/en/stable/api.html`
[13] `https://github.com/radon-h2020/radon-defuse`)

5
A Large Empirical Study on Defect Prediction in Infrastructure Code[1]

This chapter reports our experimentation to investigate the role of Machine Learning classifiers and a broad set of product metrics and process metrics for predicting failure-prone scripts. The *goal* is to evaluate the framework for IaC Defect Prediction proposed in the previous chapter in a within-project setup using supervised Machine Learning techniques, with the *purpose* of early detecting defects in IaC scripts and learning features that characterize them from the individual projects considered. The *quality focus* is on *evaluating which classification techniques and features the framework and its implementation should rely on to achieve the highest detection accuracy.* The *perspective* is of researchers who want to evaluate *in-vitro* the effectiveness of defect prediction applied to Infrastructure as Code.

The framework assessment led to the definition of several research questions:

RQ$_1$ How the classifier selection impacts the performance of Machine Learning models to predict the failure-proneness of IaC scripts?

RQ$_2$ How is the prediction performance affected by choice of the metric sets?

RQ$_3$ Which metrics are good defect predictors? That is, what are the most selected predictors and their combinations?

RQ$_1$ aims to identify the effect that the choice of machine learning classifiers has on prediction performance. First, we gathered a comprehensive and meaningful set of

[1]This chapter was published in: **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. *Transactions on Software Engineering, 2021.*

failure-prone IaC scripts and metrics to implement and assess different classifiers for predicting the failure-proneness of an IaC script. Then, we compared their performance and focused on Random Forest as the best-performing model. The contribution is a *set of classifiers suitable for detecting suspicious failure-prone IaC scripts.*

RQ_2 aims to identify the effect that the choice of metric sets – i.e., product and process metrics and groups thereof – has on the prediction performance. Indeed, product and process metrics have been widely adopted in defect prediction of systems written in general purpose languages, and in many cases, process metrics over-performed product metrics in classification performance (Moser et al. 2008; Rahman and Devanbu 2013). Nevertheless, little is known about the prediction power of both sets of metrics across domain-specific languages such as IaC.

Finally, RQ_3 aims to identify and rank the measures that highly affect prediction performance. A recursive feature selection method is performed to find the optimal number of features and to rank them according to their importance for the prediction. The contribution is *a set of metrics for detecting failure-prone IaC scripts* that developers and researchers can use to understand further and assess the quality of IaC scripts.

We evaluated the framework by analyzing the performance of five Machine Learning techniques on 104 open-source Ansible-based projects. Results show high prediction performance, with a median AUC-PR of 0.93 and an MCC of 0.80. Results also indicate that, at least for our data, product metrics identify defective IaC scripts more accurately than process metrics.

Contribution. The contribution of this chapter is a large empirical study to compare Machine Learning classifiers and metric sets to detect failure-prone IaC files. Specifically, the contribution provides (i) the empirical evidence that within-project defect prediction of Infrastructure-as-Code based on Machine Learning can reach high performance; (ii) the empirical evidence that product metrics can be more effective predictors than process metrics for within-project defect prediction of IaC; (iii) a comprehensive dataset of publicly available failure-prone Ansible files, source code measurements calculated upon these files, and their manual validation; finally (iv) pre-trained models across hundreds of projects that can be used for future studies and comparisons.

We share the datasets and models as a baseline for DevOps and the research community to better understand failure-prone IaC scripts and enable the comparison between competing approaches for defect prediction focusing on IaC.[2]

5.1. Study Setup

This section presents the methodology followed throughout the study, consisting of three phases: data collection, classifier selection, and model validation.

[2]Links to supplemental material are available on the online Appendix at `https://github.com/stefanodallapalma/TSE-2020-05-0217`

IaC ratio	Core contributors	Comments ratio	Commit frequency	Issue Frequency	CI	License	Code lines
62	406	30	421	607	21	267	136
(6%)	(39%)	(3%)	(40%)	(58%)	(2%)	(25%)	(13%)

Table 5.1: Number of projects discarded for each criterion.

5.1.1. DATA COLLECTION

We searched all the code repositories on GitHub containing Ansible code since 2014, with at least two releases and a push event to their default branch in the last six months.[3] Ansible has been developed since 2012, and we assume that two years is a reasonable amount of time for a new language to gain popularity. The total number of repositories related to our search query was 1050, and 850 were discarded from the dataset after applying the criteria depicted in Table 4.1 (Chapter 4) through the REPOSITORY SCORER. Table 5.1 shows the number of repositories discarded by each criterion. Note that the order of the criteria is not binding. Therefore, the sum of the repositories discarded by each policy is greater than the total number of repositories discarded. It is worth mentioning that the *Issue Frequency* criterion is zero when an organization uses private or external issues tracker. In that case, we manually investigated the repository to determine whether to select it for experiments. Similarly, we applied the same process when all criteria were satisfied but the *Continuous Integration* or *License Availability*.

We then ran the IAC REPOSITORY MINER on the remaining 200 repositories. During its validation, we manually removed commits deemed false positives. We also removed commits that corrected typos in comments or messages, task names, and lint warnings such as deprecation. At this stage, 61 repositories ended up with no defect-fixing commits, and therefore they were discarded, leading to 139 repositories.

Quartile	Cumulative # repos	Min	Mean	Std	Median	Max
1st	33	1	4	2	3	8
2nd	69	9	17	6	18	27
3rd	103	29	52	21	42	99
4th	139	101	314	367	175	1658

Table 5.2: Statistics on the number of defective instances for the 139 repositories containing defect-fixing commits, after applying the criteria in Table 4.1 (Chapter 4).

Finally, we selected the 106 repositories in the last three quartiles of Table 5.2, in which defective instances range from 9 to 1658. The rationale is that repositories in the first quartile have three defective instances. They would not practically allow to define a within-project model: even the best data balancing technique would have problems generating artificial instances representative of the minority class (Batista et al. 2004).

Of the 106 repositories, two failed during training. Therefore, the following research questions are assessed on the remaining 104 repositories, whose statistics are depicted in Table 5.3.

[3]Starting from March 2020.

	Releases	IaC ratio	Core contributors	Comments ratio	Commit frequency	Issue frequency	Code lines
Mean	51	67%	5	10%	26	1.50	7585
Std	77	15%	4	8%	86	2.93	14892
Min	2	19%	2	1%	2	0.01	165
Median	29	68%	4	8%	8	0.67	2570
Max	589	97%	18	45%	863	23.97	109565

Table 5.3: Statistics of the 104 analyzed repositories according to the inclusion criteria, and the last three quartiles in Table 5.2. *License* and *Continuous integration* are not shown as boolean and *true* for all the repositories.

5.1.2. Classifier Selection

We relied on *five* classification algorithms, namely Naive Bayes (NB), Logistic Regression (LR), Classification And Regression Trees (CART), Random Forest (RF), and Support Vector Machine (SVM), as they have been widely used for defect prediction (Hosseini et al. 2019). On the one hand, Naive Bayes and Logistic Regression are simple to understand and interpret and are fast learners, as they do not require much training data. On the other hand, Decision Tree, Random Forest, and Support Vector Machine are more flexible and powerful, as they can fit many functional forms and do not make assumptions about the underlying function. Although they are less efficient at training time due to many parameters, they can provide high-performance models. Furthermore, they also have a good level of interpretability. Decision Tree and Random Forest can be analyzed by observing the trees' path and the decision nodes to understand which feature affected the final decision. Support Vector Machine provides a formula that is a weighted sum over features. The value of each feature can be analyzed to identify the extent to which it contributed to classifying the given instance. The five models complement each other in terms of pros and cons while keeping a reasonable degree of explainability and are good candidates for our goals. All of the above techniques were implemented using the scikit-learn framework in Python.[4]

5.1.3. Model Validation

To compute the model performance, we reported the following evaluation measures, as we believe they are the most suitable for imbalanced data sets where one class is observed more frequently than the other class:

- **Area Under the Curve - Precision-Recall (AUC-PR).** This measure summarizes the precision-recall (PR) curve. On imbalanced or skewed data sets, PR curves are a valuable alternative to ROC curves to highlight the performance differences lost in ROC curves (Goadrich et al. 2006; Boyd et al. 2013). Please consider that we used AUC-PR to tune the models when applying cross-validation.

- **Matthews Correlation Coefficient (MCC).** This measure focuses on the quality of binary classifications. A coefficient of +1 represents a perfect prediction, 0 is no better than a random prediction, and -1 indicates total disagreement between prediction and observation (Matthews 1975).

[4]https://scikit-learn.org/stable/supervised_learning.html

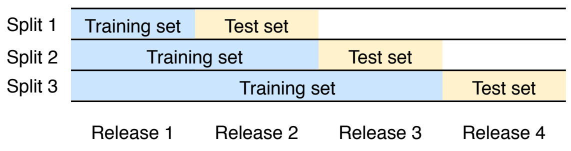


Figure 5.1: Walk-forward release cross-validation.

The performance is analyzed in terms of mean and standard deviation. A more comprehensive table containing all the evaluation measures for each project can be found in the online Appendix.[5]

We evaluated the suitability and performance of the proposed classifiers by training them with different configurations of normalization, data balancing, and hyper-parameters. The model selection was guided by a randomized search of models' parameters through a *walk-forward validation* (Falessi et al. 2020). In walk-forward, the dataset represents a time series that can be divided into sets ordered chronologically, e.g., a project releases. In each run, the selected set represents the test set, and all data preceding it represents the train set; this way, the test set is prevented from having data antecedent to the train set. Afterward, the model performance is computed as the average among runs. The number of iterations is equal to the number of parts minus one. Specifically, we trained each model on the first n releases and tested the (n+1)-th release for every integer $n \in [1, |Releases|)$. This process is illustrated in Figure 5.1.

5.2. IMPACT OF LEARNING TECHNIQUES

The framework presented in Chapter 4 allowed us to gather a comprehensive and meaningful set of data to implement a learning-based method for predicting the failure-proneness of an IaC script, as discussed throughout this section, and answer the following research question:

How the classifier selection impacts the performance of Machine Learning models to predict the failure-proneness of IaC scripts?

5.2.1. METHODOLOGY

We created 104 Machine Learning models (one per project) trained and tested on different configurations. At this step, we used all 108 metrics as features and decided, on purpose, not to use any feature selection algorithm as we were interested in analyzing the capabilities of the prediction models using the whole set of metrics and considered only releases having at least one defective script.

First, we analyzed the number of times each evaluated classifier achieved the best performance (i.e., it was the best model in terms of AUC-PR) 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 significance of the differences. Specifically, for both AUC-PR and MCC, we reported *p-values* from a matched

[5]Available online at https://github.com/stefanodallapalma/TSE-2020-05-0217/blob/master/METRICS.md

Learning technique	Occurrences
Random Forest	98
Support Vector Machine	26
Logistic Regression	17
Decision Tree	16
Naive Bayes	1

Table 5.4: Number of times a model appears among the best-performing models.

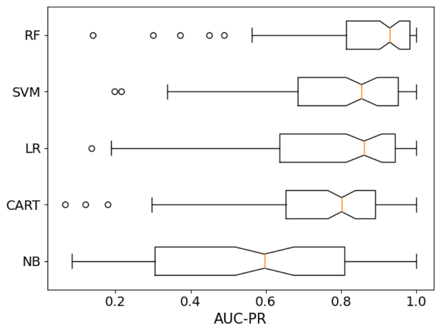
pair Wilcoxon's rank test (Wilcoxon 1992) for all pairs of techniques, along with the *effect size* using Cohen's *d* (Cohen 1988). Wilcoxon's rank test determines whether two or more sets of pairs are different from one another in a statistically significant manner, while Cohen's *d* measures the effect size for the comparison between two means. Cohen's *d* is considered negligible below 0.2; small between 0.2 and 0.5; medium between 0.5 and 0.8; large above 0.8 (Cohen 1988). Finally, given the number of comparisons being performed, we set the significance level to $\alpha = 0.01$ and performed a posthoc Bonferroni's correction (Weisstein 2004). Specifically, we adjusted the significance level according to the number of comparisons (i.e., ten); therefore, $\alpha = 0.001$. Finally, we reported several statistics (i.e., mean, median, minimum, maximum, standard deviation) for each evaluation measure for the classifier achieving the best performance.

5.2.2. Quantitative Results

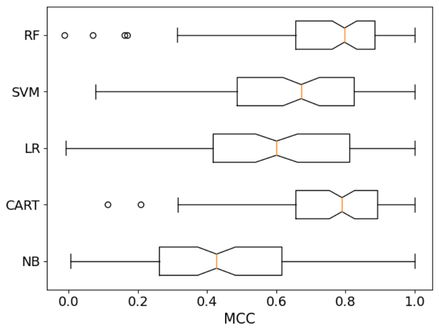
Table 5.4 shows each evaluated model's occurrences as the best model for any given project in terms of AUC-PR. As can be observed, *Random Forest* is the classifier that occurs most (98/104), followed by *Support Vector Machine* (26/104), *Logistic Regression* (17/104), and *Decision Tree* (16/104). *Naive Bayes* has been observed among the best models only once. It is worth mentioning that the sum of the occurrences is not equal to 104 as, for some projects, multiple models achieved the same performance. In particular, we observed the co-occurrences (*CART, LR, RF, SVM*) four times, (*LR, RF, SVM*) three times, and the co-occurrences (*CART, LR, NB, RF, SVM*), (*CART, RF, SVM*), (*CART, LR, RF*), and (*RF, SVM*) once.

Furthermore, Figure 5.2a - Figure 5.2b and Table 5.5 - Table 5.6 show that the mean differences among the learning techniques in AUC-PR and MCC are generally high and of practical significance. The average PR area ranges from 0.56 for the worst-performing technique, namely *Naive Bayes*, to above 0.87 for the best-performing techniques, namely *Random Forest*. *Random Forest* is the learning method with the lowest standard deviation and thus yields the most stable results regardless of the metrics used; the minimum is right below 0.14, the maximum is 1.00, and the standard deviation 0.16.

More specifically, Figure 5.2a and Figure 5.2b show a clear overview of the performance and their differences. According to Chambers 2018, although not a formal test, the non-overlap of two boxes' notches indicates strong evidence (95% confidence) that their medians differ. Thus, we observed that the median performance of *Random Forest* differs from the remaining techniques. In contrast, the notches' overlap between *Support Vector Machine*, *Logistic Regression*, and *Decision Tree* suggests that their median performance does not differ significantly. The only exception is *Decision Tree*'s



(a)



(b)

Figure 5.2: (a) Area under the precision-recall curve and (b) Matthews correlation coefficient of each learning technique. The models trained using *Random Forest* perform statistically better than those relying on the remaining classifiers, both in AUC-PR and MCC. Follows, in order, *Support Vector Machine*, *Logistic Regression*, and *Decision Tree*, albeit their performance difference is negligible. The only exception is that the *Decision Tree*'s MCC is very close to *Random Forest*, with no significant difference.

	Random Forest	Decision Tree	SVM	Logistic Regression	Naive Bayes
Random Forest	–	Medium	Small	Medium	Large
Decision Tree	**-12**	–	Small	Negligible	Medium
SVM	**-8**	5	–	Negligible	Large
Logistic Regression	**-10**	**3**	-2	–	Large
Naive Bayes	**-31**	**-19**	**-23**	**-21**	–

Table 5.5: Statistical comparison of mean *AUC-PR* among learning techniques. Values below the diagonal are the differences between pairs of techniques (in %, significant in bold) – negative value means that the model in the row performed worse than the one in the column. Labels above the diagonal are the effect size.

	Random Forest	Decision Tree	SVM	Logistic Regression	Naive Bayes
Random Forest	–	Negligible	Small	Medium	Large
Decision Tree	2	–	Medium	Large	Large
SVM	**-9**	**-12**	–	Small	Medium
Logistic Regression	**-15**	**-16**	-5	–	Medium
Naive Bayes	**-28**	**-30**	**-18**	**-13**	–

Table 5.6: Statistical comparison of mean *MCC* among learning techniques. Values below the diagonal are the differences between pairs of techniques (in %, significant in bold) – a negative value means that the model in the row performed worse than the one in the column. Labels above the diagonal are the effect size.

MCC, which is similar to *Random Forest* and significantly better than the remaining techniques. Significant differences between pairs of techniques are shown in bold in Table 5.5 and Table 5.6. Differences in AUC-PR between *Decision Tree* and *Support Vector Machine* (5%) and between *Support Vector Machine* and *Logistic Regression* (2%) are not statistically significant (*p-value* > 0.001). The values above the diagonal indicate the effect size. The largest effect size occurs between *Random Forest* and *Naive Bayes* with a magnitude of 1.40, meaning that the difference between the two means is larger than one standard deviation. The effect size between *Random Forest* and *Support Vector Machine* (0.44) and between *Support Vector Machine* and *Decision Tree* (0.23) is small. The effect size between *Decision Tree* and *Logistic Regression* (0.12) and between *Support Vector Machine* and *Logistic Regression* (0.09) is negligible.

Likewise, the average MCC ranges from 0.46 for the worst-performing technique, namely *Naive Bayes*, to approximately 0.75 for the best-performing technique, namely *Random Forest*. The latter is also the learning technique with the lowest standard deviation (~0.21) along with *Decision Tree* (~0.18); the minimum is around 0, while the maximum is 1.00. Regarding MCC, the differences between *Random Forest* and *Decision Tree* (2%) and *Support Vector Machine* and *Logistic Regression* (5%) are not statistically significant. Similarly, the respective effect sizes (i.e., 0.09 and 0.21) are negligible and small.

5.2.3. Qualitative Results

The performances are generally very high, though we observed 20 projects where all the models perform poorly, with a median AUC-PR of 0.45 (against the 0.90 of the remaining). Looking closer at these projects' characteristics and their inclusion criteria in Table 4.1 (Chapter 4), we observed that the models are trained on a relatively lower number of releases (a median of 13 against the 32 for the projects with good models' performance). Because the models were trained and validated through a walk-forward validation across releases, a higher number of releases allows more runs to train and validate the model to select the one that maximizes the AUC-PR. Furthermore, the median ratio of defective instances to the total instances differs by 13% between the two groups of projects: 7% and 20%, respectively. The lack of defective instances in the first group might have impacted the models' capabilities to discriminate between failure-prone and neutral instances, thus leading to bad performance.

Looking at the metrics, we observed a significant difference for the metric *ChangeSetMax* (a median of 16 in the first group compared to 4 in the second), meaning that the 20 projects with poor models' performance have a median maximum number of files committed together of 16. Although our analysis considers only Ansible scripts, it is possible that in those projects, the presence of many Ansible files committed together led to an imprecise identification of failure-prone scripts for the sole reason that a fixing-commit modified them along with the actual failure-prone file(s).

Concerning the differences among the models, although the difference between *Support Vector Machine* and *Logistic Regression* is tiny, we observed 11 projects with a highly variable AUC-PR (i.e., a median difference of ~25%). In those projects, the former model outperformed the second 8 times. In those cases, the project's characteristics were similar. However, we observed that the median number of commits was higher (14) than

	AUC-PR	MCC	Precision	Recall	F1
Mean	0.87	0.74	0.77	0.84	0.77
Std	0.16	0.21	0.21	0.16	0.20
Min	0.14	-0.01	0.00	0.00	0.00
Median	0.93	0.80	0.82	0.89	0.82
Max	1.00	1.00	1.00	1.00	1.00

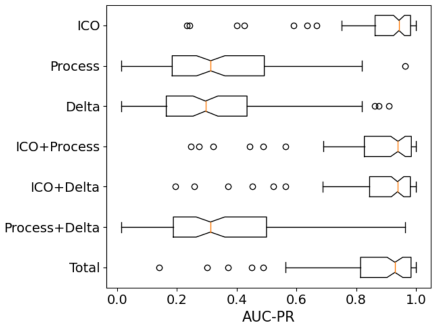
Table 5.7: Performance statistics of Random Forest across the 104 repositories.

the opposite (8). Similarly, the median number of instances used for training was almost three times higher in projects where *Support Vector Machine* performed better than *Logistic Regression*, while the percentage of defective instances to the total number of instances was approximately the same (~7%). That might suggest that *Logistic Regression* exploited better the projects with a small amount of data, but that might have been negatively affected by noise in the data present in larger projects. However, it is unclear how these attributes affected the models' performance. Also, the median *ChangeSetMax* was notably lower in the first case (10) than in the second case (25). Similarly, the median *ChangeSetAvg* (i.e., the average number of files committed together) was 3 in the first case and 4 in the latter, suggesting that in those projects, SVM is less deteriorated by the noise in the dataset compared to LR. It is worth mentioning that, regardless of its importance for the final classification of IaC scripts' failure-proneness, the *ChangeSet* metric suggests that the dataset might contain several false positives, in the presence of which some models might perform worse than others.

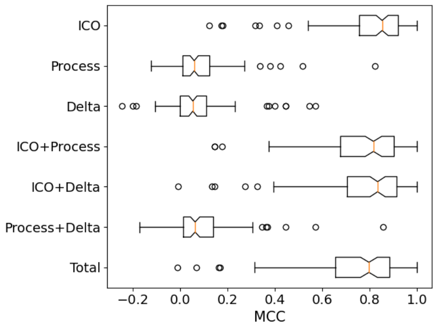
Finally, *Random Forest* outperforms the other models, regardless of the metrics used or the project's characteristics. Therefore, according to our findings, we analyzed the performance of *Random Forest*, i.e., the best performing classifier. Table 5.7 reports the average performance for all projects concerning PR area, precision, recall, F1, and MCC. Although the minimum AUC-PR is 0.14, the mean and median are high: 0.87 and 0.93, respectively, with a standard deviation of 0.16, meaning that, on average, models' AUC-PR range between 0.71 and 1. Although the standard deviation might seem high, the coefficient of variation (CV) is only 0.18. As a rule of thumb, a $CV \geq 1$ indicates a relatively high variation, while a $CV < 1$ is considered low. In our case, the high average MCC (0.74) indicates a strong agreement between the predictions and the observed values.

RQ_1 Summary

The models trained using Random Forest perform statistically better than those relying on the remaining classifiers. Moreover, the difference is statistically different with a large effect size.



(a)



(b)

Figure 5.3: (a) Area under the precision-recall curve and (b) Matthews correlation coefficient of each metric set. Models featuring IaC-oriented metrics perform statistically better with a large effect size than those not relying on them, both in AUC-PR and MCC. Adding process, delta metrics, or both, does not significantly affect their performance. Models featuring process and delta metrics alone, or their combination, perform poorly with almost no difference.

5.3. Effect of Metric Sets

Using more features might lead to better performance due to the increased quantity of information that a Machine Learning technique can exploit. Nevertheless, at the same time, we are aware that using the whole set of metrics might decrease the detection performance as more features could increase noise. Furthermore, extracting all features could not be feasible in some projects or even cause more work in the metric collection. Therefore, in RQ_2, we aim to understand the effect of different *types of metrics* on the prediction performance:

How is the prediction performance affected by choice of the metric sets?

5.3.1. Methodology

We evaluated the relative predictive power of the metric sets presented in Section 4.3 (*ICO*, *Delta*, and *Process*) by combining data to construct seven candidate metric sets: *ICO*, *Delta*, *Process*, *ICO* + *Delta*, *ICO* + *Process*, *Delta* + *Process*, *Total*. Then, as for RQ_1, we reported several statistics and performed statistical analysis to compare the magnitude of the differences between the metric sets. In particular, for each evaluation measure, we showed the results of the matched pair Wilcoxon's rank test for all pairs of metrics sets aggregated across the best learning method from **RQ**$_1$, i.e., *Random Forest*, along with the *effect size* computed using Cohen's *d*. Given the number of comparisons we performed, we set the significance level to $\alpha = 0.01$ and applied the posthoc Bonferroni's correction.

	ICO	ICO + Delta	ICO + Process	Total	Process + Delta	Process	Delta
ICO	–	Negligible	Negligible	Negligible	Large	Large	Large
ICO + Delta	-1	–	Negligible	Negligible	Large	Large	Large
ICO + Process	**-2**	**-1**	–	Negligible	Large	Large	Large
Total	**-2**	-2	0	–	Large	Large	Large
Process + Delta	**-55**	**-54**	**-53**	**-53**	–	Negligible	Negligible
Process	**-55**	**-55**	**-54**	**-53**	0	–	Negligible
Delta	**-58**	**-58**	**-56**	**-56**	**-3**	**-3**	–

Table 5.8: Statistics of mean *AUC-PR* among metric sets. Values below the diagonal are the differences between pairs of metric sets (in %, significant in bold). A negative value means that the model featuring the row's metrics performs worse than a model featuring the column's metrics. Labels above the diagonal are the effect size.

	ICO	ICO + Delta	ICO + Process	Total	Process + Delta	Process	Delta
ICO	–	Negligible	Negligible	Small	Large	Large	Large
ICO, Delta	**-3**	–	Negligible	Negligible	Large	Large	Large
ICO, Process	**-3**	-1	–	Negligible	Large	Large	Large
Total	**-6**	**-4**	-3	–	Large	Large	Large
Process, Delta	**-71**	**-7**	**-68**	**-65**	–	Negligible	Negligible
Process	**-72**	**-72**	**-69**	**-66**	-1	–	Negligible
Delta	**-73**	**-73**	**-69**	**-66**	-1	-1	–

Table 5.9: Statistics of mean *MCC* among metric sets. Values below the diagonal are the differences between pairs of metric sets (in %, significant in bold). A negative value means that the model featuring row's metrics performs worse than a model featuring the column's metrics. Labels above the diagonal are the effect size.

5.3.2. Results

Figure 5.3a - Figure 5.3b and Table 5.8-Table 5.9 show that the mean differences among the metric sets, in terms of AUC-PR and MCC, are generally high and of practical significance. The average PR area ranges from 0.32, for the *Delta* metric set up to 0.90 for the *ICO* metric set consisting solely of Infrastructure as Code features.

The *ICO* metrics are also the metrics with the lowest standard deviation and thus yield the most stable results; the minimum is right above 0.23, the maximum is 1.00, and the standard deviation 0.14. These results align with those observed in **RQ**$_1$, suggesting that the results were due to those metrics. Indeed, there is a significant performance decrease (shown in bold in Table 5.8) when adding other metric sets to the *ICO* or using them alone. Like **RQ**$_1$, this can be caught at a glance from the boxplot analysis in Figure 5.3a and Figure 5.3b. First, the gap between the performance of models featuring *ICO* and those relying on *Process* or *Delta* metrics is evident. Second, the overlap of notches between *ICO*, *ICO+Process*, *ICO+Delta*, and *Total* suggests that the median performance does not differ significantly among models using those features. Consequently, adding process, delta metrics, or both to the IaC-oriented metrics does not significantly affect the model's performance.

For example, adding the *Delta* or the *Process* metrics significantly decreases the performance by 1% and 2% on average. However, the effect size is negligible. Therefore, those metrics do not contribute to improving the results. At the same time, although they do not worsen the performance significantly, using them would require an additional effort for metrics collection. Furthermore, using the *Total* set of metrics decreases the performance similarly (i.e., 6% on average). Finally, using *Process* and *Delta* metrics alone produces poor results, with a median AUC-PR of 0.31.

Likewise, the average MCC ranges from 0.07 for the worst metric set, *Delta*, to above 0.80 for the best metric set, *ICO*. The *ICO* metrics lead to a model with the third lowest standard deviation (~0.18, after the 0.13 of *Process* and *Delta* metrics); the minimum *ICO*'s MCC is right above 0.12 while the maximum is 1.00, and its median is 0.92. Similar to AUC-PR, the difference between the model trained on the *ICO* metric set is significantly higher than the ones trained on the other metric sets.

RQ$_2$ Summary

The models with IaC-oriented metrics perform statistically better than those relying on the remaining metric sets. Moreover, the difference is statistically significant with a large effect size.

5.4. Best Predictors

In the previous section, we observed that models trained using product metrics outperform those using process and delta metrics. While the previous research question aimed to study how the prediction performance is affected by choice of the metrics as grouped by their type, the following research question aims to identify and rank *individual metrics* based on their impact on the prediction performance:

Which metrics are good defect predictors? That is, what are the most selected predictors and their combinations?

5.4.1. Methodology

To answer this research question, we performed a recursive feature selection to find the metrics that maximize the performance and to rank them according to their importance for the prediction. Given an external estimator that assigns weights to features (e.g., the importance of each feature in a Random Forest model), Recursive Feature Elimination (RFE) recursively considers smaller and smaller sets of features that optimize the performance criteria. The algorithm trains the estimator on the initial set of features and ranks the features by importance. Then, the least important features are pruned from the current set. The procedure is recursively repeated on the pruned set until the algorithm eventually reaches the desired number of features to select. However, RFE requires selecting the number of features to keep, which is often not known in advance. Therefore, to find the optimal number of features, we need to apply cross-validation to score the different feature subsets and select the best collection of features based on that score. To this end, we used the *RFECV* method available in *scikit-learn*[6] along with the *Random Forest* model from RQ$_1$ as estimator and *walk-forward* as cross-validation.

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

Rank	Predictor	Type	Occur.
1	NumTokens	ICO	84
1	TextEntropy	ICO	84
1	LinesCode	ICO	84
2	NumKeys	ICO	78
3	AvgTaskSize	ICO	64
4	LinesBlank	ICO	58
5	NumParameters	ICO	57
6	NumUniqueNames	ICO	55
7	NumDistinctModules	ICO	54
8	NumConditions	ICO	49

(a)

Rank	Predictor	Occur.
1	NumTokens, LinesCode	79
2	NumTokens, TextEntropy	76
3	LinesCode, TextEntropy	75
4	LinesCode, NumKeys	72
4	NumKeys, TextEntropy	72
4	LinesCode, NumTokens, TextEntropy	72
5	NumKeys, NumTokens	71
6	LinesCode, NumKeys NumTokens	69
7	LinesCode, NumKeys, TextEntropy	68
8	NumKeys, NumTokens, TextEntropy	67

(b)

Table 5.10: (a) Ten most recurring features that maximize the AUC-PR; (b) Ten most recurring combinations of features that maximize AUC-PR.

5.4.2. Result

The results of RFECV show a median of 11 optimal features per model, with a mean test PR area and standard deviation of 0.89 and 0.14, respectively. The average AUC-PR aligns with the results observed in RQ_1. However, the lower number of optimal features suggests that most of them are redundant and decrease the overall performance.

Table 5.10a and Table 5.10b rank the ten most-recurring features and their combinations. *NumTokens*, *TextEntropy*, and *LinesCode* are the three features that occur most among the features selected by the RFECV. It is interesting to note that the most recurring process metrics, namely *ChangeSetMax*, *ChangeSetAvg*, and *CodeChurnCount*, only occur 42, 30 and, 29 times, respectively, and are at rank 11, 17, and 18; while the most occurring delta metric is *DeltaTextEntropy* at rank 20 with 27 occurrences, and most of them are between rank 26 and 42 (i.e., the last rank).

RQ_3 Summary

IaC-oriented metrics tend to maximize the prediction performance. In particular, NUMTOKENS, TEXTENTROPY, *and* LINESCODE *are the most occurring predictors.*

5.5. Discussion and Limitations

In RQ_1, we found out that the collected metrics have a high prediction power for the failure-proneness of IaC. Regardless of the metrics used, *Random Forest* provided the best results, over-performing other learning techniques, reaching a median AUC-PR of 0.93, an MCC of 0.80, and an F1-score of 0.82. We observed that the performance difference between *Support Vector Machine* and *Logistic Regression* and between *Logistic Regression* and *Decision Tree* is not statistically significant, although the former provided better results most of the time. Consequently, depending on the desired model flexibility and the available computational resources, one can choose them interchangeably without negatively affecting the prediction.

The results achieved in the context of RQ_2 and RQ_3 are surprising. In RQ_2, we found that models trained on process metrics have poor performance compared with code metrics. In RQ_3, we observed that the top 13 predictors (in terms of occurrences among the most critical features resulting from the recursive feature elimination) include IaC-oriented metrics only. Therefore, we conclude that structural code metrics outperform process metrics for the collected Ansible-based projects, although the latter is often more effective when predicting the failure-proneness of source code instances in traditional defect prediction (Moser et al. 2008; Rahman and Devanbu 2013). We conjecture that this result is due to the fewer number of infrastructure code changes than application code, limiting the information exploitable by the process metrics.

5.5.1. Implications for Research and Practice

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 core contributors, size of commits, lines of code, and the ratio of IaC files). This aspect is particularly interesting for 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.

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 RQ_3. This aspect will reduce the number of features to collect, reduce the number of tools required to mine them, and speed up the training phase. In addition, for each project on which we trained our models, we report several statistics such as the size, number of commits, and core contributors to allow practitioners to compare their projects with those used in this study and use our pre-trained models.

5.5.2. Threats to Validity

This section describes the threats that can affect the validity of our study.

Threats to Construct Validity

Threats to *construct validity* concern the relation between the theory and the observations, and in this work are mainly due to the repositories we collected and the measurements we performed. Such a threat is the most occurring in our study and relates to:

- *Imprecise identification of relevant repositories*: it is possible that some repositories relevant for the analysis are missing or that some repositories are not relevant (e.g., too small for practical significance). However, we filtered out repositories based on ten criteria to identify active, IaC-related, and well-engineered software projects to mitigate this threat.

- *Imprecise identification of the failure-prone scripts*: some scripts might have been incorrectly identified or not identified as “failure-prone”. To mitigate this threat, we considered the developers’ intent (i.e., by analyzing issue labels) and selected a state-of-the-art strategy to analyze the commit messages. Although using commit messages to mine defect information might be biased, Rahman, Posnett, et al. 2013 reported that increasing the sample size can leverage the possible bias in defect data. We also ignored renamed files without modification to the source code, added or removed by the defect-fixing commits. Besides, failure-prone scripts were extracted from a set of ~5,000 defect-fixing commits. Although our selection reflects a statistically-relevant set for our available data, we acknowledge this may not be the complete population of publicly available defect-fixing commits. We confer the validity of our study to the systematicity of the approach utilized for data collection, which warrants reasonable certainty that a large part of the available fixing-commits was considered.

- *Undocumented bugs in the system*: we relied on the issue tracker to identify bugs fixed during the change history. Undocumented bugs (i.e., no issues related to the bug) could be present in some scripts, mistakenly classifying failure-prone scripts as "neutral". To mitigate this threat, we considered the commit messages.

- *Approximations due to identifying fix- and defect-introducing changes using keywords and the SZZ algorithm*: we relied on an ad hoc set of issue labels, keywords to identify defect-fixing commits, and the SZZ implementation available in PyDriller to identify defect-introducing commits. A diverse combination of labels, keywords, and the SZZ algorithm may bring different results. Nevertheless, we selected common, representative keywords used by developers to indicate fixes (while analyzing commit messages). Furthermore, we collected the labels related to bugs by manually analyzing the labels of each project.

- *Imprecise computation of process and code metrics*: poor implementation of the metrics might lead to the imprecise computation of their values, thus making the predictor wrongly believe a metric is more important than others. To mitigate this threat, we implemented the metrics following their documentation and adopting a test-driven approach (Beck 2003). First, we documented the metrics with the intended behavior and examples. Then, we implemented the test cases beforehand the production code to improve our confidence in the metric miner.

Threats to Internal Validity

Threats to *internal validity* concern internal factors we did not consider that could affect the investigated variables. In particular, the choice of the metrics (e.g., product vs. process metrics) might positively or negatively influence the classification. We mitigated this threat by considering a comprehensive set of metrics gathered from the literature comprising

- Metrics that take into account the structural properties of infrastructure code,

- Metrics that track the changes between two successive releases for each of the structural metrics, and

- Metrics that consider the development process rather than the structural characteristics of the code.

The nature of data could hinder the accuracy of the models. In this context, multicollinearity is a condition where two or more independent variables are highly correlated. Although this aspect is particularly relevant for *Logistic Regression*, we did not consider in order to guarantee a fair comparison with the other models. However, our results suggest that the multicollinearity effect could be significant. To assess this, we have conducted an additional analysis to reduce the multicollinearity by discarding features with a Variable Inflation Factor (VIF) greater than 10 (Neter et al. 1996).[7] Our results show that the models where we apply VIF have a median MCC and AUC-PR of 21% and 10% lower than those not including it; albeit, the effect's magnitude of these differences is negligible. Nevertheless, the slight difference might indicate that our methodology did not influence the results much.

Similarly, data balancing is a critical aspect of defect prediction. Class imbalance and the proposed framework's balancing techniques could affect the model's performance. Although we observed that *no balancing* is the balancing technique occurring the most (followed by random over-sampling and random under-sampling), we acknowledge that the impact of other state-of-the-art balancing techniques (e.g., SMOTE) should be investigated as well. Nevertheless, exhaustive analysis of the impact of feature selection and balancing techniques is out of scope for this study: a careful evaluation of such aspects would require dedicated studies, which we plan as future research.

Threats to External Validity

Threats to *external validity* concern the generalization of results. First, we analyzed 104 Ansible-based systems from different application domains with different characteristics, such as the number of contributors, commits, and size. However, systems from different ecosystems (e.g., Chef, Puppet) and orchestration language (e.g., TOSCA) should be analyzed with the same framework to corroborate our findings. Our framework can be easily extended to other configuration management and orchestration languages. In particular, process metrics are language agnostic and can be extracted from any versioning control system regardless of the language. Many of the structural metrics we collected can also be extended to other languages, given their general-scope nature (e.g., *NumTokens* and *TextEntropy*).

Second, our work revolves around within-project defect prediction, and, as such, we aim at learning features that characterize failure-prone IaC scripts from the individual projects considered. Repositories having no defects or a small number of defective instances were not used in this context: indeed, the absence of defects would not allow any machine learner to distinguish failure-prone from neutral scripts, while projects having a tiny amount of defective instances would not practically allow the definition of a within-project model because even the best data balancing technique would have problems in generating artificial instances that are representative of the minority class (Batista et al. 2004). For these reasons, our framework might not generalize to projects with either zero or small amounts of defective instances. Instead, those projects would be enforced to use a cross-project strategy, where information is gathered from external projects so machine learners could use it in their environments. Nevertheless, the investigation of the proposed framework performance in cross-project defect prediction

[7] Available on the online Appendix.

is part of our future research.

Finally, another threat is related to classifier selection. We chose five classifiers widely used in previous studies on bug prediction (e.g., Bowes et al. 2018; Di Nucci, Palomba, De Rosa, et al. 2018).

Threats to Conclusion Validity

Threats to *conclusion validity* concern the relation between the treatment and the outcome. The metrics used to evaluate our defect prediction approach (i.e., AUC-ROC, precision, recall, F-Measure, and MCC) are widely used in the evaluation of the performance of defect prediction techniques (Hall et al. 2012; Fu, Menzies, and Shen 2016; Tantithamthavorn et al. 2016; Di Nucci, Palomba, De Rosa, et al. 2018; Hosseini et al. 2019). Moreover, we used AUC-PR as an alternative and more conservative measure than the AUC-ROC to evaluate performance for highly imbalanced problems.

Furthermore, since we needed to exploit change-history information to compute the metrics we collected, 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.

5.6. Summary

This chapter presented an approach for within-project defect prediction of Infrastructure-as-Code based on product and process metrics. The approach enables the analysis of infrastructure code and detection of defects at the implementation level.

Although the empirical study targets Ansible, we believe the models may be transferred to other languages (Chef, Puppet), as many of the most recurring features are general-purpose metrics.

We compared five machine learning methods for defect prediction: Decision Tree, Logistic Regression, Naive Bayes, Random Forest, and Support Vector Machine. We have crafted a dataset of publicly available Ansible-based IaC scripts extracted from relevant and active GitHub repositories to train the models. Furthermore, we classified scripts as failure-prone or neutral and then analyzed the predictive power of the models. Results show that Random Forest outperforms other models, with a median AUC-PR = 0.93, and product metrics outperform process metrics.

6
An Empirical Study on Blob Blueprint in TOSCA[1]

This chapter focuses on lousy coding practices, which reveal another harm in software development: symptoms of wrong style usage or a lousy design known as code smells. Code smells do not directly cause system failures but violate best practices and design principles, negatively affecting the readability and code maintainability (Fowler 2018).

Code smells are broadly researched for traditional source code development (Sharma and Spinellis 2018), as their detection can enhance software quality. In the scope of IaC, Guerriero et al. 2019 investigated the state of practices in adopting IaC using the data from 44 semi-structured interviews with senior developers. They observed that large IaC scripts, referred to as Blob blueprint, occur among the most common bad practices in the industry when developing infrastructure code.

Unfortunately, only a few works exist on IaC code smells (Rahman, Mahdavi-Hezaveh, et al. 2019), and they focus on specific technologies and languages, such as Puppet or Ansible. However, Guerriero et al. 2019 identify as one of the best practices “recombining diverse formats by abstraction using the OASIS TOSCA standard for IaC and including multiple formats inside node-type definitions”. Indeed, technology-agnostic infrastructure code, such as the OASIS Topology and Orchestration Specification for Cloud Applications[2] (TOSCA) can build upon existing configuration and orchestration languages to improve the readability and portability of configuration files across platforms (Binz et al. 2014; Lipton et al. 2018). It enables automated deployment of technology-independent and multi-cloud compliant applications, managing applications, resources, and services regardless of the underlying cloud platform or infrastructure. From a business viewpoint, TOSCA *“expands customer choice, reduces cost,* and *increases business agility* across

[1]This chapter was published in: **S. Dalla Palma**, C. van Asseldonk, G. Catolino, D. Di Nucci, F. Palomba, and D. A. Tamburri. *“Through the Looking Glass...”* An empirical study on blob infrastructure blueprints in the Topology and Orchestration Specification for Cloud Applications. *Journal of Software: Evolution and Process, 2022.*

[2]`https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca`

the application life cycle. The synergy between these benefits *accelerates overall time-to-value*" (Lipton et al. 2018).

What is more, the fourth and sixth authors of the paper are active members of the OASIS TOSCA Standard Technical Committee and used this presence to enact discussions about this work in some of the committee's meetings. As a result, practitioners shared concerns about "long, blob-like blueprints". In particular, they mentioned that the most critical hazards connected to infrastructure abuse come from lousy coding practices concerning the security of IaC blueprints. The targets for such practices are mainly found in long, *blob-like* blueprints. Such issues reportedly can yield irreparable infrastructure damage as well as loss or even theft of data as much as leaking of industrial secrets to a point in which manual inspection of long blueprints is required.[3]

In this chapter, we conjecture that identifying code smells in technology-agnostic infrastructure code and related metrics opens up opportunities for (1) building a general, automated service continuity quality model designed explicitly for configuration orchestration languages and (2) understanding how lousy coding practices affect service infrastructure continuity (Baresi et al. 2020).

This chapter aims to identify structural code measures that characterize complex blueprints and analyze the effectiveness of candidate metric- and unsupervised learning-based techniques in detecting those blueprints. Motivated by this goal, it aims to address the following research questions in the context of TOSCA:

RQ_1 To what extent can structural code metrics distinguish between Blob and sound blueprints?

RQ_2 To what extent can metric-based and unsupervised-learning-based techniques detect Blob blueprints?

RQ_3 What metrics are the most effective to maximize the performance of those detectors?

We conducted a case study involving over 700 blueprints and a prominent traditional implementation code smell: *Large Class* or *Blob*. The smell is one of the most frequently investigated for general-purpose application code (Sobrinho et al. 2018) and the most frequently mentioned in infrastructure code in the industry (Guerriero et al. 2019); Guerriero et al. 2019 referred to it as *Blob blueprint*, namely a too large IaC script – heron, we use the same nomenclature.

We selected this smell for its implementation ease, frequency, and potential impact on infrastructure code quality: it highly impacts fault- and change-proneness (Li and Shatnawi 2007; Khomh et al. 2012; Palomba, Bavota, Di Penta, et al. 2018) and can potentially be observed and easily implemented in TOSCA.

We built upon the studies by Sharma, Fragkoulis, et al. 2016 and Schwarz et al. 2018 and analyzed traditional structural code metrics on IaC smell detection to corroborate their findings on a technology-agnostic language (i.e., TOSCA), and to investigate the extent to which metric- and unsupervised learning-based techniques can be leveraged to detect Blob blueprints.

[3]https://www.oasis-open.org/2022/06/06/emerging-compute-models-recommendations-and-sample-profile-v1-0-published-by-tosca-tc/ (Section 2.1)

The motivation behind focusing on metric-based and unsupervised-learning-based detectors is many-fold: Metrics-based smell detectors are the most frequent and easy to implement; they calculate a set of metrics, such as *lines of code*, *coupling*, and *cohesion*, upon the source code and detect smells if they exceed a given threshold (Marinescu 2005). Nevertheless, determining a suitable threshold is a non-trivial challenge. On the other hand, unsupervised-learning-based detectors overcome shortcomings such as determining threshold values and might significantly reduce the effort of collecting and identifying smelly blueprints, similarly to previous work on software defect prediction (Fu and Menzies 2017; Li, Shepperd, et al. 2020).

Contribution. The contribution of this chapter is an empirical study to compare metric- and unsupervised learning-based techniques to detect *Blob blueprints*. The study provides insights on structural code metrics that corroborate previous findings; empirically shows that metric-based detectors are more accurate than unsupervised learning-based detectors; and provides a comprehensive dataset of publicly available TOSCA blueprints, including source code measurements calculated upon these blueprints and manually validated observations related to *Blob blueprints*.[4]

6.1. THE BLOB BLUEPRINT

In general-purpose application code, *Large Classes* contain too many fields and methods and rely on several external data classes, making them low cohesive (Fowler 2018; Sharma and Spinellis 2018). In technology-agnostic infrastructure code, Blob blueprints are too large or complex topologies that should be modularized further.

Despite not being addressed in the literature, Sharma, Fragkoulis, et al. 2016 and Schwarz et al. 2018 investigated the similar *Insufficient Modularization* smell in Puppet and Chef, respectively, and provided three detection rules:

1. Configuration files that contain more than one class (in Chef) or resource (in Puppet); or
2. Class declarations that are too large (more than 40 lines of code); or
3. Class declarations that are too complex (max nesting depth more than three).

In TOSCA, (node and relationship) types and templates are analogous to abstract and concrete classes (Binz et al. 2014); hence, they can be considered a leading indicator for Blob blueprints – the total number of types and templates maps rule 1.

It is accepted that the larger a module (a blueprint in this context), the more difficult it is to comprehend; hence, the topology's length can be considered a leading indicator for Blob blueprints – the number of code lines maps rule 2.

Unfortunately, rule 3 could not be easily mapped: it is unclear how to define complexity in TOSCA and measure it due to its novelty and difference compared with general-purpose programming languages. In general, a complexity measure tries to capture the difficulty in understanding a module (a blueprint in this case). For example, Sharma et

[4]https://github.com/jade-lab/tosca-smells

al. and Schwarz et al. measured the complexity of an IaC script through its maximum nesting depth – e.g., in *if* statements. Unfortunately, this is impossible in TOSCA because of its declarative nature, so we had to identify a different complexity measure.

Following the definition of *Large Class* for general-purpose application code, we computed the number of *interfaces* and *properties* as analogous to the number of methods and attributes and the well-known *lack of cohesion of methods* (LCOM) (Chidamber et al. 1994) to measure cohesion. In particular, the latter measures the number of connected components in a class – a connected component is a set of methods that access the same class-level variables. Ideally, there should be only one connected component in each class. Unfortunately, the same metric cannot be used for infrastructure code due to its different structure and characteristics; thus, for Blob blueprints, we defined a *connected component* as a set of related types or templates (rather than methods in application code) and blueprint-level properties (rather than attributes in application code). Then, similarly to Sharma, Fragkoulis, et al. 2016, we used the following algorithm to compute LCOM in a blueprint:

1. Consider each *node* and *relationship template* defined in the topology template as a node in a graph. Initially, the graph contains the disconnected components (DC) equal to the number of templates.

2. Identify the *parameters* and the *variables* declared in the topology template.

3. For each parameter and variable, identify the components using them; then, merge them into a single component.

4. Compute the lack of cohesion as $LCOM = |DC|$.

Listing 9 shows a TOSCA topology template with two *node templates* – `Docker` and `marathon` – that access two disjoint groups of input each: *mem_size* and *num_cpu* for `Docker`; *rclone_user* and *rclone_password* for `marathon`. Therefore, the topology consists of two connected components composed of one node; hence, LCOM = 2.

In addition to cohesion, we computed the *number of imports* to measure the *efferent coupling* (a.k.a. fan-out), which defines “the number of components on which a particular component depends. Components with a high efferent coupling value are sensitive to the changes introduced to their dependencies. Moreover, the deficiencies of their dependencies naturally manifest themselves in these components.”[5]

Please note that we first relied on our knowledge of “complexity” in application code to elicit a set of relevant metrics in TOSCA. Then, we performed preliminary non-structured interviews within the OASIS TOSCA Technical Committee to understand their view on the matter and evaluate the extent to which our definition of blobs and infrastructure complexity match. In particular, practitioners were asked to map complexity characteristics to blob-like measurements and infrastructure blueprint characteristics; two key complexity aspects emerged that still require dedicated attention even beyond the scope of this study:

[5]https://www.entrofi.net/coupling-metrics-afferent-and-efferent-coupling/

Listing 9 An example of how the measure LCOM can be interpreted and computed in TOSCA. The two nodes in the topology (`Docker` and `marathon`) access two disjoint groups of input each: *mem_size* and *num_cpu* for `Docker`, and *rclone_user* and *rclone_password* for `marathon`. Therefore, LCOM = 2.

```
tosca_definitions_version: "tosca_simple_yaml_1_0"

# Imports, description, and metadata...

topology_template:
  inputs:
    mem_size:
      # ...
    num_cpus:
      # ...
    rclone_user:
      # ...
    rclone_password:
      # ...

  node_templates:
    Docker:
      type: "tosca.nodes.indigo.Docker"
      cababilities:
        host:
          properties:
            mem_size: { get_input: mem_size }
            num_cpus: { get_input: num_cpus }

    marathon:
      type: "tosca.nodes.indigo.Marathon"
      properties:
        command: { get_input: run_command }
        environment_variables:
          RCLONE_CONFIG_USER: { get_input: rclone_user }
          RCLONE_CONFIG_PASS: { get_input: rclone_password }
```

1. **Blob blueprints reflect a lower bound for complexity.** Blueprints that implement multiple modules, interface hooks, and dependencies tend to develop maintainability problems and vulnerability to infrastructure penetration or chaos (Basiri et al. 2016).

2. **Automated testing Blob IaC is nigh impossible.** A considerable number of negative characteristics are sparsely related to automated testing (e.g., low code understandability, low code reuse); this warrants the necessity of defining operationally multiple and fine-grained complexity measures to be used in a combined complexity function for further automation.

While this study focuses on the first of the above points, we are releasing all materials and automation borne of this study to encourage further replication of our computational results and further research on the matter. Finally, we refined and implemented the considered metrics based on their input.

6.2. STUDY SETUP

This section sets up the empirical study, which consists of five phases: data collection, data preparation (exploratory analysis and data pre-processing), detectors building, and performance evaluation and comparison.

The goal is to investigate the role of metric-based and unsupervised-learning-based detectors to identify Blob blueprints and to extend DEFUSE to support infrastructure code smell detection. The *perspective* is of both researchers and practitioners. The former is interested in assessing, through *in-vitro* experimentation, the effectiveness of metric- and unsupervised learning-based code smell detection applied to technology-agnostic infrastructure code. The latter is interested in evaluating how unsupervised learning can be used in practice.

6.2.1. Data Collection

TOSCA is a novel standard; hence, we mined GitHub using the search query `tosca` to collect a comprehensive set of 636 repositories. Afterward, we discarded repositories without releases because interested in blueprints considered functioning and collected all the `.tosca` files – i.e., the blueprints – from the last release of each project.

In addition, TOSCA blueprints can also have a *.yml* extension. Hence, we searched for YAML files containing a `tosca_definitions_version`: that keyword identifies the versioned set of normative TOSCA type definitions to validate those types defined in the TOSCA Simple Profile, and it is mandatory.[6]

Finally, we discarded blueprints used for testing or examples (i.e., those containing *test* or *example* in the filepath), as they are not representative of production blueprints targeted by this study; the dataset at this stage consists of 1036 blueprints from 42 repositories.

6.2.2. Data Preparation

The blueprints thus collected were scanned using the Python library *toscametrics*[7] to extract the metrics defined in Section 6.1 to create the dataset for experiments; 287 blueprints were discarded because of syntax errors (123) or duplicates (164), leading to a dataset of 749 blueprints.

From that dataset, two authors manually annotated a statistically-relevant set of 290 blueprints sampled to have an acceptable margin of error of 5% and a 95% confidence level to create the ground truth for the exploratory analysis described below and compare techniques. The annotation was performed *before* the subsequent analyses so that we could avoid additional bias; furthermore, the inspectors actively discussed their operations multiple times to convey a decision in order to reduce subjectivity.

The two assessors scanned each resource and labeled it as smelly or sound based on their experience and understanding of the blueprint's semantics. The authors have at least *four* years of experience in code quality and IaC research. In addition, the fourth author is an active member of the OASIS TOSCA Technical Committee;[8] as such, he participates in monthly meetings where the Committee discusses with TOSCA practitioners about status and challenges of the language.

Each blueprint was subjectively analyzed considering the overall length, the number of nodes and relationships, and their scope based on type, description, properties, and

[6] https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html
[7] https://github.com/radon-h2020/radon-tosca-metrics
[8] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca

interfaces; these criteria were defined based on the theoretical model in Section 6.1 before annotating. We also considered complexity in terms of difficulty in understanding the operations performed by the blueprint. A web application was developed and shared among the assessors to facilitate the annotation.[9]

Then, the assessors calculated Cohen's Kappa (Cohen 1960) to measure their agreement. Cohen's Kappa ranges between 0 and 1, with 0 indicating no agreement between the two raters and 1 indicating perfect agreement. In case of disagreements, the assessors met and discussed the disagreed blueprint to convey a decision. The resulting ground truth consisted of 248 sound and 42 blob blueprints – after reaching a complete agreement in the resolution phase from an initial Cohen's Kappa of 0.56 (i.e., moderate agreement).

EXPLORATORY ANALYSIS

We tested all metrics separately using statistical analysis before detecting Blob blueprints. For each metric, we applied the non-parametric Mann–Whitney U test (Conover 1998) with a significance level $\alpha = 0.01$ to measure if the distributions of this metric within Blob and sound blueprints differ significantly. We adjusted the significance level to $\alpha = 0.002$ to control for the randomness of the observations using Bonferroni's correction (Weisstein 2004) according to the number of comparisons (i.e., five); thus, the two groups differ significantly (for the considered metric) if $p < 0.002$. It is worth noting that a metric distributed differently does not necessarily distinguish Blob and sound blueprints. However, these results provide some hints as to why a machine learning approach that combines these features can be successful.

Beyond the p-value interpretation, we used Cliff's delta (Cliff 1993) to measure the effect size, i.e., the magnitude of the difference between two populations. Cliff's delta ranges from zero to one: according to Kampenes et al. 2007, a value below 0.147 is considered trivial; between 0.147 and 0.33, it is small; between 0.33 and 0.474, it is medium, and it is large above 0.474.

PRE-PROCESSING

This section describes how the metrics were pre-processed for experimentation. First, we normalized data as a common requirement for many machine learning estimators. Typically this is done by removing the mean and scaling to unit variance. However, outliers can often influence the sample mean or variance negatively;[10] therefore, we resorted to the `RobustScaler` available in *scikit-learn*: it scales data similarly to the min-max normalization but uses the interquartile range instead of the max-min range to be outliers-robust, applying the formula below:

$$\frac{x - Q_1(x)}{Q_3(x) - Q_1(x)}$$

As for feature selection, the metrics were initially intended for code smell detection. However, some metrics may correlate to others; this is a problem in unsupervised learning, as the concept they represent gets a higher weight than other concepts, leading the

[9]Demo accessible at `https://smell-annotator.web.app/` using the token: 9WhBsZe1EiDhFgVXtBPn

[10]`https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html`

final model to be skewed toward that particular concept, which might be undesirable. For that reason, we controlled for multicollinearity through the Variable Inflation Factor (VIF) (Mansfield et al. 1982) by discarding the features having a value larger than 10, a widely-used rule-of-thumb (Catolino et al. 2020). In addition, we determined the optimal set of features for building the detectors through a stepwise forward selection. More specifically, all the metrics but those already selected are tested against the Matthew Correlation Coefficient (MCC) at each step; the metric that significantly improves MCC the most is added to the optimal set.

6.2.3. Detectors Building

The pre-processed data and the selected features are used to build the metrics- and unsupervised learning-based smell detectors described below.

Metrics-based detectors

A metric-based detector takes source code as the input, calculates a set of source code metrics that capture the characteristics of a given smell, and detects that smell by applying a suitable threshold to one or more metrics (Sharma and Spinellis 2018). In most cases, setting the threshold values is a highly empirical process, and it is guided by similar past experiences and hints from the author of the metrics Marinescu 2004. For example, as mentioned in Section 6.1, Sharma, Fragkoulis, et al. 2016 and Schwarz et al. 2018 detect a similar smell called Insufficient Modularization if a configuration script contains more than 40 lines of code or an abstraction contains more than one class or define. Nevertheless, those thresholds do not hold to TOSCA because of the differences between these languages. Indeed, blueprints rarely contain a single type because of their nature, and types are usually small. In addition, no previous works on smell detection for TOSCA exist; hence, no hints are available from past experiences. Nevertheless, statistical techniques can define a suitable threshold for each metric in those cases. In this case, we used the *Interquartile Rule* used in previous works (Aniche et al. 2018) as a baseline:

$$T(x) = Q_3(x) + 1.5 \times IQR(x)$$

The formula defines a threshold using the third quartile ($Q_3(x)$) and the interquartile range ($IQR(x) = Q_3(x) - Q_1(x)$) to spot blueprints representing upper outliers for a specific metric x – i.e., smelly instances.

Typically, a threshold is calculated for each metric, and a rule combines them through logic operators: The *logic `OR`* detects smells if *any* metric exceeds the respective threshold and is used in this study. The *logic `AND`* detects smells if *all* metrics exceed the respective thresholds but might be impractical as the probability of detecting smelly instances drops when the number of metrics increases. For this reason, we relied on multivariate outliers to consider multiple metrics at once.

The standard method for multivariate outlier detection uses the Mahalanobis distance McLachlan 1999; Filzmoser et al. 2004 – a measure of the distance between a point p and a distribution D. The Mahalanobis distance measures how many standard deviations the point p is far from the mean of D. The distances are typically interpreted by comparing the corresponding χ^2 value (with the degrees of freedom equal to the number

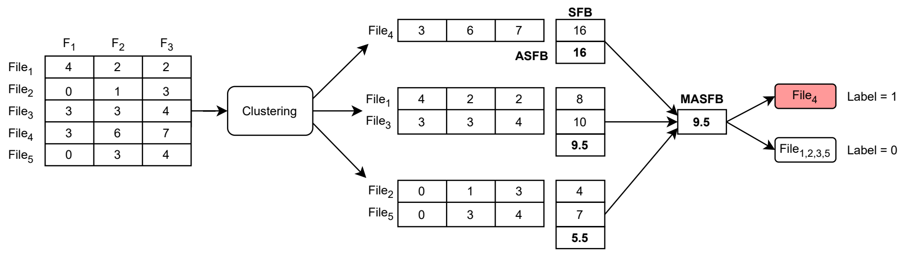


Figure 6.1: The cluster labeling scheme. Once clusters are defined, the feature values of blueprints in each cluster are summed (SFB). Then, the average SFB (ASFB) is calculated for each cluster. To support scenarios with multiple clusters (> 2), the median values of these ASFBs (MASFB) are calculated so that clusters where $ASFB > MASFB$ are labeled as smelly and the remaining as sound.

of variables) to a cut-off p-value. Cases with a *p-value* < .001 are likely to be considered outliers.[11]

UNSUPERVISED LEARNING-BASED DETECTORS

The unsupervised learning detectors proposed in this study revolve around four popular clustering techniques available in the Python framework *scikit-learn* (Pedregosa et al. 2011), namely *KMeans*, *AgglomerativeClustering*, *MeanShift*, and *BIRCH*. Not only did we rely on these techniques for their popularity,[12] but also because they have already been used in the context of code smell detection for traditional source code (Azeem et al. 2019; Guggulothu et al. 2019). Furthermore, we used the implementations by *scikit-learn* to ensure an easy operationalization for practitioners and replication for researchers. A detailed description of these techniques can be found in the scikit-learn's official documentation.[13]

Number of Clusters. Most of the algorithms mentioned above require the number of clusters to be specified in advance. We could not know that information beforehand; therefore, we resorted to the Silhouette coefficient (Rousseeuw 1987) to validate the goodness of a clustering technique and select an appropriate number of clusters. The Silhouette coefficient ranges between -1 and +1: a coefficient close to +1 indicates that the objects are well matched to their cluster and poorly matched to neighboring clusters. A value close to -1 indicates too many or too few clusters, whereas a coefficient close to 0 indicates overlapping clusters. Therefore, we performed a randomized search on different hyper-parameters configurations for every clustering technique and retained the configuration that maximized the Silhouette coefficient across ten runs.

Cluster Labelling. The clusters resulting from the previous step have unique identifiers from a technical standpoint; they are not labeled as smelly or sound. Zhang et al. 2016 proposed a heuristic to label clusters in the context of defect prediction that Xu et

[11]https://www.statisticssolutions.com/univariate-and-multivariate-outliers/
[12]https://dataaspirant.com/unsupervised-learning-algorithms/
[13]https://scikit-learn.org/stable/modules/clustering.html

al. 2021 adapted to support scenarios with more than two clusters. Figure 6.1 depicts the heuristic that we instantiated for smell detection by changing the labels as follows:

1. **S**um up the **f**eature values of **b**lueprints in each cluster (SFB).
2. Calculate the average SFB for each cluster (ASFB).
3. Calculate the median of these ASFBs (MASFB).
4. Label every observation in each cluster as *smelly* if $ASFB > MASFB$; *sound* otherwise.

Step 4) assumes that Blob blueprints generally have larger values than sound blueprints for the considered metrics. Although this reasoning was proposed for defect prediction D'Ambros et al. 2012c; Nam et al. 2015; Zhang et al. 2016, we argue that it applies to the smell we investigated. Because of step 4), in case one cluster is generated, all the observations are labeled as sound.

6.2.4. Performance Evaluation

We evaluated the performance of each technique on the ground truth in terms of *Precision* and *Recall* (Baeza-Yates et al. 1999), defined as follows:

$$precision = \frac{TP}{TP+FP}$$

$$recall = \frac{TP}{TP+FN}$$

Where TP is the number of smelly instances classified correctly; TN denotes the number of non-smelly instances classified correctly; FP and FN measure the number of classes for which the model fails to identify the smelliness of classes by declaring these classes as smelly (FP) or non-smelly (FN).

Then, we computed the Matthews Correlation Coefficient (MCC) (Matthews 1975), a regression coefficient that combines all four quadrants of a confusion matrix, thus also considering true negatives. Its formula is:

$$MCC = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

TP, *TN*, and *FP* represent the number of true positives, true negatives, and false positives, respectively, while *FN* is the number of false negatives. Matthews Correlation Coefficient ranges between -1 and +1:

- +1 indicates a perfect prediction.
- 0 suggests that the model is no better than a random one.
- -1 indicates total disagreement between predictions and observations.

Metric	Mean Blob	Mean others	U	Effect size
LinesCode	669	79	7381*	Large
NumTypesAndTemplates	15	3	6675*	Large
LCOM	12	2	6631*	Large
NumInterfaces	6	1	5303*	Medium
NumProperties	64	9	6638*	Large
NumImports	6	2	4699	Small

Table 6.1: Overview of the features and results of the statistical analysis of the metrics. Mann–Whitney U test significant (*) if $p < 0.002$ (corresponding to a non-corrected $p < 0.01$ for each test).

Please note that these measurements are primarily used in classification evaluation, but, as this research applied a binary clustering with known ground truth, they are applicable. In addition, we calculated the adjusted Rand index (ARI) – a common metric for clustering evaluation. It measures the similarity between the ground truth assignments and the clustering approach assignments on the same samples.

Finally, for each algorithm, we reported whether their MCC differs significantly. To this end, we applied the non-parametric Mann–Whitney U test Conover 1998 with a significance level $\alpha = 0.01$. To better control for the randomness of our observations, we used Bonferroni's correction Weisstein 2004 to adjust the significance level according to the number of comparisons (i.e., 15). Thus, the results are significant at the significance level $\alpha = 0.001$. P-values lower than this value show that the MCC of the two algorithms differs significantly.

Beyond the p-value interpretation, we calculated the effect size using Cliff's delta Cliff 1993 to measure the magnitude of the difference between two populations and ranges from zero to one. For example, according to Kampenes et al. 2007, a value below 0.147 is considered trivial; between 0.147 and 0.33, it is small; between 0.33 and 0.474, it is medium, and it is large above 0.474.

Please note that we evaluated the techniques across 100 experiments to gain insights into performance variability. across 100 experiments. Each experiment considered a perturbed version of the original data set consisting of at least 290 uniformly sampled observations without replacement (i.e., a statistically relevant sample size) and reported statistics like median, mean, and standard deviation. We generated these versions upfront to evaluate all the techniques on the same data sets.

6.3. RESULTS

Table 6.1 shows a statistical evaluation of the metrics, and Figure 6.2 depicts their distribution through violin plots: Blob and sound blueprints distributions differ significantly for all considered metrics compared with the ground truth; however, *NumImports* is distributed independently from whether the blueprint is smelly or not, with a small effect size. Overall, this first analysis hints that their use, and the combination thereof, is a good proxy for detecting Blob blueprints. Therefore, we used those metrics to build the smell detectors.

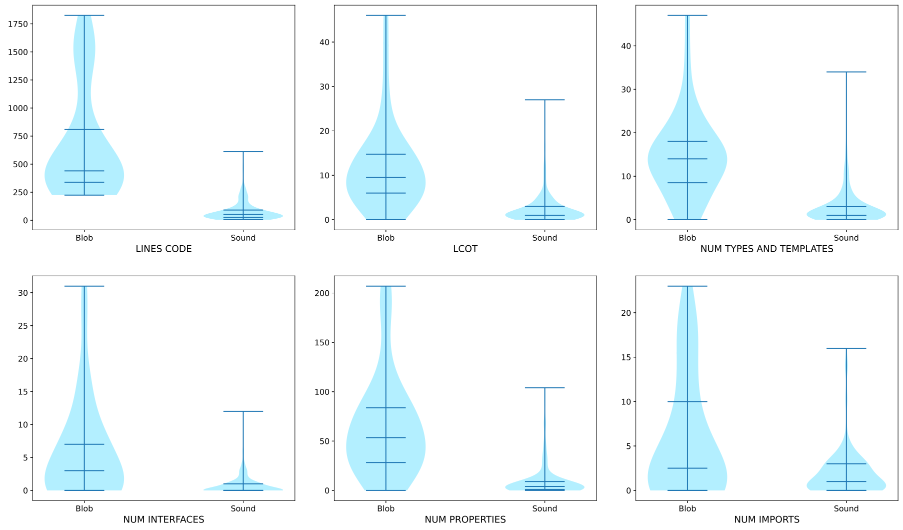


Figure 6.2: Results of the statistical analysis of metrics distribution: the violin plots show that the population of Blob and sound blueprints have different distributions for every considered metric compared with the ground truth.

RQ$_1$ Summary

All the considered structural code metrics, except NumImports, can distinguish between Blob and sound blueprints with medium to large effect sizes.

Table 6.2 shows the experiment results for each detector, and Figure 6.3 compares detectors' performance in terms of MCC through violin plots.

Metric-based detectors maximize MCC; their mean ranges from 0.60 using the Mahalanobis distance to 0.80 using the interquartile rule. The latter yields the most stable results regardless of the metrics used; the minimum is below 0.73, the maximum is 0.85, and the standard deviation is 0.03. Metric-based detectors also maximize F1 with a mean of 0.63 and 0.80.

MeanShift - the best among unsupervised learning-based detectors - performs like metric-based detectors, although with a 22% worse MCC than the best metric-based detector, with a large effect size. While, AgglomerativeClustering - the worse among unsupervised learning-based detectors - maximizes precision but drops recall quickly.

Table 6.3 shows that the differences among the detectors are generally high and of practical significance; e.g., all detectors worsen the best detector's MCC by 22% to 35% with a large effect size. However, most of them perform moderately well in MCC and F1.

These results are encouraging because false positives and false negatives are minimal. Indeed, false detections might highly worsen code quality: On the one hand, false positives falsely alarm for smells that do not exist - if this happens too often, developers could stop trusting the smell detector. On the other hand, false negatives falsely reassure

Algorithm	MCC		Precision		Recall		F1		ARI		FN	FP	TN	TP
	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std				
IQR	**0.79**	0.03	0.68	**0.05**	**0.99**	**0.02**	**0.80**	0.04	–	–	0	8	140	20
MeanShift	0.63	0.06	0.63	0.17	0.77	0.21	0.65	0.07	**0.52**	0.08	5	14	142	17
Mahalanobis	0.60	**0.02**	0.76	0.06	0.54	0.04	0.63	**0.02**	–	–	10	3	146	11
KMeans	0.59	0.06	0.59	0.10	0.72	0.12	0.64	0.07	0.51	**0.07**	6	11	143	16
BIRCH	0.56	0.09	0.64	0.14	0.63	0.20	0.59	0.13	0.48	0.11	9	8	148	13
Agglomerative	0.51	0.10	**0.82**	0.08	0.38	0.14	0.49	0.13	0.42	0.12	14	2	147	7

Table 6.2: Performance statistics across 100 perturbed versions of the dataset.

	IQR	MeanShif	Mahalanobis	KMeans	Birch	Agglomerative
IQR	–	Large	Large	Large	Large	Large
MeanShift	**-22**	–	Large	Medium	Large	Large
Mahalanobis	**-24**	**-5**	–	Small	Large	Large
KMeans	**-25**	**-6**	+2	–	Small	Large
Birch	**-29**	**-11**	**-7**	**-5**	–	Medium
Agglomerative	**-35**	**-19**	**-15**	**-14**	**-9**	–

Table 6.3: Statistical comparison of MCC among detectors. The values below the diagonal are the differences between pairs of techniques in % (significant in bold). A negative value indicates that the detector in the row performed worse than the one in the column. The labels above the diagonal are the effect size.

the lack of code smells - if this happens too often, developers could become less vigilant during code reviews.

RQ$_2$ Summary

Metric-based detectors perform statistically better than unsupervised learning-based detectors. The difference is statistically different with a large effect size.

Finally, Figure 6.4 shows how metrics impact performance. During the stepwise forward selection described in Section 6.2.2, we tracked the MCC, Precision, and Recall variation when adding metrics at each step. The process adds the metric that maximizes the MCC – or minimizes the drop in MCC – at each step. For example, the IQR-based detector (top-left graph) maximizes MCC when only LinesCode is used. Then, adding NumImports decreases the MCC, although the combination *(LinesCode, NumImports)* maximizes the MCC among pairs of metrics that include *LinesCode*. Similarly, adding *LCOM* – the best among the remaining metrics – further decreases the MCC.

Figure 6.4 also shows that performance varies visibly across metrics sets and detectors. For example, the number of code lines maximizes MCC for the IQR-based detector; simultaneously, performances are minimal for the Mahalanobis-based detector. As for the unsupervised learning-based detectors, for the same metric, MCC increases for MeanShift but decreases for the remaining; precision tends to increase or stay constant; the opposite applies to recall.

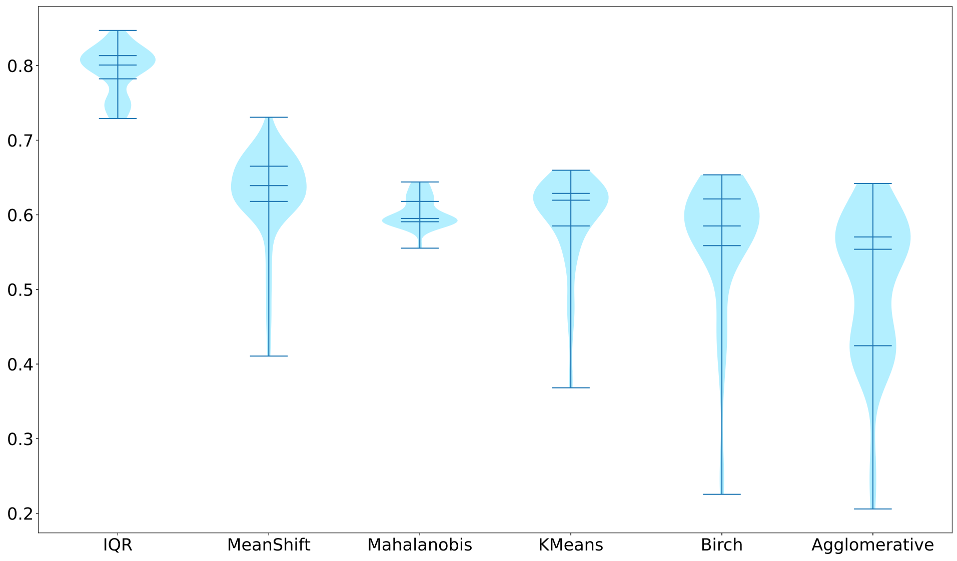


Figure 6.3: Matthews correlation coefficient across metric- and unsupervised learning-based detectors. The detector built using the *Interquartile rule* performs statistically better than those relying on unsupervised learning.

Algorithm	Selected features
IQR	*LinesCode*
MeanShift	*NumTypes, LinesCode, LCOM, NumProperties*
Mahalanobis	*NumInterfaces, LinesCode, LCOM, NumProperties*
KMeans	*NumInterfaces, NumTypes*
Birch	*NumInterfaces, NumTypes, LCOM*
Agglomerative	*NumInterfaces*

Table 6.4: Features that maximize MCC for each detector.

In conclusion, Figure 6.4 suggests that all the metrics improve performance; hence, they should be regarded together while building detectors. Besides, analyzing all combinations was infeasible for the sake of time. Nevertheless, we observed several metrics that recur across the optimal set of selected features, shown in Table 6.4: among them, the number of interfaces – analogous to the number of methods in traditional programming – occurs the most.

RQ$_3$ Summary

The number of interfaces appears to be a leading metric to maximize the overall performance. The number of types and templates, code lines, and LCOM follows.

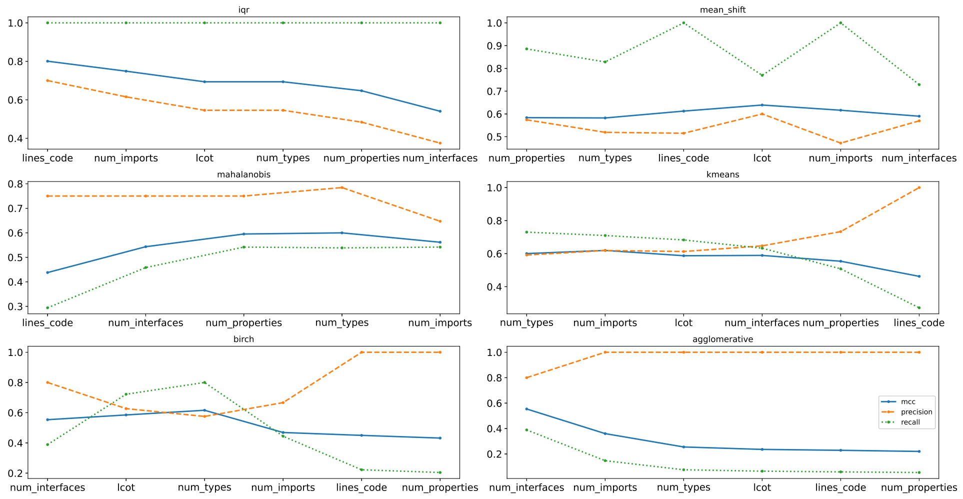


Figure 6.4: Stepwise forward selection to track the change in performance when adding the metric that maximized MCC – or, minimized its drop – at each step. For example, the IQR-based detector (top-left graph) maximizes MCC when only *LinesCode* is used. Then, adding *NumImports* decreases the MCC, although the combination (*LinesCode*, *NumImports*) maximizes the MCC among pairs of metrics that include *LinesCode*. Similarly, adding *LCOM* – the best one among the remaining metrics – further decreases the MCC.

6.4. DISCUSSIONS

6.4.1. THREATS TO VALIDITY

This section describes the threats that can affect the validity of this study.

THREATS TO CONSTRUCT VALIDITY

Threats to *construct validity* concern the relation between the theory behind the executed methodology and the found observations by assessing whether the observed outcome corresponds to the effect we think we are measuring.

The developed source code measurements may not appropriately represent the intended characteristic. We mitigated this threat by using two approaches. First, we searched for measures that have been empirically validated multiple times for traditional code smell detection that could be ported to TOSCA; then, we consulted the official TOSCA documentation to identify possible measures related to the blueprint complexity and size. Afterward, we employed the test-driven development approach (Beck 2003) to implement the intended measurement and improve it until all the initial unit tests passed.

Another threat relates to the construction of the ground truth, done manually. As manual work, such as labeling, can be prone to human errors, we acknowledge this as a possible threat to construct validity. We tried to mitigate this threat by summarizing the most prevalent definitions found in the literature for the analyzed smell. Furthermore, the authors of this study performed the validation, which poses a threat to the construction validity due to the bias regarding the perception of what metrics or quality attributes characterize the smell; involving external experts would mitigate this bias. However, it is worth noting that the authors involved in the validation have multiple years of experience in code quality and IaC research; although the annotators were not external, they could still be considered experts enough for this task. Finally, the annotators performed the validation *before* the subsequent analyses to mitigate additional bias; they also actively discussed their operations multiple times to reduce subjectivity.

Finally, the GitHub repositories were automatically crawled based on a search string; hence, relevant repositories may miss due to a too conservative search string.

Internal Validity

Threats to *internal validity* concern the possibility that other factors could cause the outcome but were not measured during the research.

A possible threat to internal validity is the source code measurement selection: some unimplemented source code measurements could significantly influence the results. We tried to mitigate this risk by selecting measurements that have been empirically validated multiple times for traditional code smell detection.

External Validity

Threats to *external validity* relate to the generalizability of the obtained results outside the scope of the research. We observed various threats to external validity in this work.

The dataset contains only publicly available blueprints; therefore, it might not fully represent the population of TOSCA blueprints as private blueprints used in industry.

Furthermore, despite including a large subset of publicly available TOSCA blueprints, the dataset size might negatively affect the modeling performance of the clustering algorithm used.

Finally, we only used four clustering algorithms; however, such algorithms are among the most popular and intuitive and provide easy operationalization and interpretability for practitioners.

Conclusion Validity

Threats to *conclusion validity* concern the appropriate usage of statistical tests and reliable measurement procedures to ensure the quality of the conclusions.

A possible threat to conclusion validity is the implementation of the detectors techniques used and the applied valuation strategy in our work. We built metrics-based and unsupervised-learning-based detectors upon previous works Sharma, Fragkoulis, et al. 2016; Schwarz et al. 2018 and the Python framework scikit-learn, respectively. Then, the evaluation metrics used (i.e., Precision, Recall, and MCC) are widely used techniques for evaluating the performances of binary classification tasks.

Furthermore, the test used to analyze the metrics distribution and measure performance might threaten the conclusion validity: there are many statistical tests whose choice relies upon the data structure, data distribution, and variable type, and the result can differ accordingly. Therefore, we applied a commonly used non-parametric test that makes no assumptions about the data distribution to mitigate this threat. Besides, the multiple simultaneous hypothesis tests performed could produce false positives; hence, we applied Bonferroni's correction to adjust the significance level to control the probability of committing a type I error.

6.4.2. Discussions, Implications, and Lessons Learned

In **RQ**$_1$ and **RQ**$_3$, we found that traditional source code metrics, such as the number of methods (*interfaces* in TOSCA), classes (*types* and *templates* in TOSCA), code lines, and lack of cohesion are good indicators of complex blueprints when mapped to their respective concepts in TOSCA. This result corroborates, on a technology-agnostic language, the findings of Schwarz et al. 2018 and Schwarz et al. 2018.

Besides, **RQ**$_2$ shows that practitioners should prefer metrics-based detectors to unsupervised learning-based detectors. The latter helps overcome shortcomings such as determining threshold values required by the former and possibly reduce the effort of collecting and identifying smelly blueprints. We believe that, despite the performance observed in this study, unsupervised-learning-based detectors can still play a role in detecting Blob blueprints and other smells in TOSCA. However, a broader range of TOSCA blueprints and metrics may be needed to enhance them.

Implications for Researchers and Practitioners

The results above pose several implications for researchers and practitioners as described below.

- **Implications for researchers:** There is still room for research in this area, and we argue for more empirical work on configuration smells to broaden our knowledge of complex blueprints and enhance the catalog of code smells for IaC. Our findings put a baseline to investigate which metrics should be used to detect Blob blueprints. However, further research is needed to understand the relationship between the smelliness of the TOSCA code and the collected metrics. These results can lead to a better understanding of which features to utilize to improve code smell detection in TOSCA and enable the comparison of competing approaches.

- **Implication for practitioners:** Practitioners can exploit our findings and shared material to implement novel methods and tools based on a small set of features such as those elicited in this paper. These tools will warn developers of complex blueprints and ultimately help reduce technical debt. For example, we already used those metrics as a proxy to predict failure-prone TOSCA blueprints within the scope of the European project called RADON, aimed at pursuing a broader adoption of serverless computing technologies within the European software industry.[14] One of the RADON key pillars is quality assurance for TOSCA. The results from analyzing blueprints in one of our partner's use cases within the project show that they help distinguish blueprints that may induce technical debt.[15] According to them, these metrics could ensure avoiding complex code representations.

6.4.3. Lesson Learned

This section reports insights from validating complex blueprints that could benefit future researchers to identify more fine-grained complexity measures for Blob blueprints.

[14] https://radon-h2020.eu/

[15] https://radon-h2020.eu/wp-content/uploads/2021/09/D6.5-Final-Assessment-Report.pdf (Section 4.3.3.4)

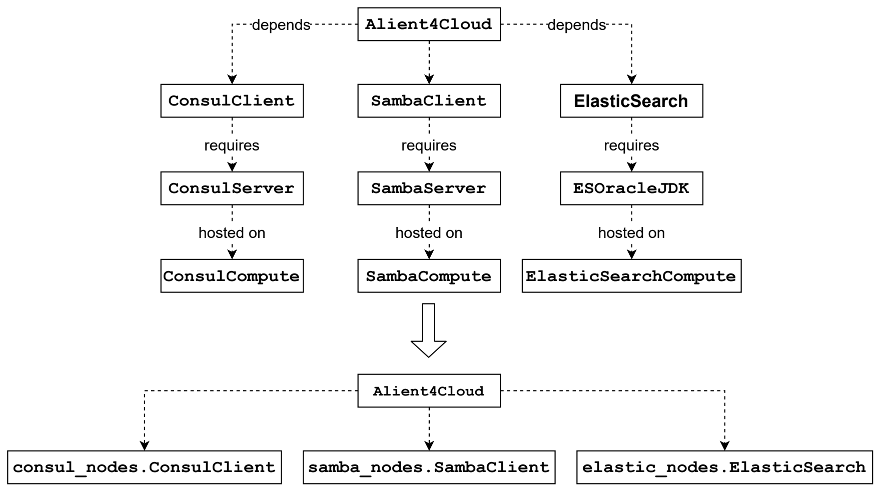


Figure 6.5: An excerpt of a Blob blueprint with nodes targeting different technologies and a possible refactoring suggestion.

REFACTOR NODES BASED ON TARGET TECHNOLOGY

While validating Blob blueprints, we observed several defining too many nodes and targeting different technologies. For example, a blueprint from *Alien4Cloud*[16] contains 16 nodes, a subset of which targets three different technologies: *Consul*, *Samba*, and *Elasticsearch*. Figure 6.5 shows those nodes and their dependencies. Therefore, it might be advisable to refactor those types in different blueprints to reduce the complexity, each grouping type targeting the same or similar technology. Then, those blueprints can be imported into the one at hand.

REFACTOR NODES BASED ON TYPES

TOSCA provides types describing the possible building blocks for constructing a service template (Caballer et al. 2018). For example, node types describe the kinds of nodes, relationship types describe possible relations among those nodes, and policy types logically group related nodes that need to be orchestrated or managed together to achieve result. While it is possible and might be advisable for a blueprint to define one or more components of each type, too many different types hinder the blueprint's clarity. Instead, those blueprints could be refactored in separate files for each type or group.

MOVE WORKFLOWS INTO THEIR SEPARATE FILES

Some blueprints have long workflows that significantly increase the size and decrease readability – workflows are used for topology (un)deployment or run-time management. Blueprints could benefit if workflows were defined in their files and imported into the blueprint.

[16]https://raw.githubusercontent.com/alien4cloud/csar-public-library/d08f5ac3f3f5279ad65fdf8c025459fafac37e75/org/alien4cloud/alien4cloud/topologies/a4c_ha/type.yml

6.5. SUMMARY

This chapter enhanced the current knowledge of current practices in Infrastructure as Code (IaC), and the detection of configuration smells. Current scientific work insufficiently addressed the characteristics of best practices within IaC (Rahman, Mahdavi-Hezaveh, et al. 2019); only a handful of previous works investigated configuration smells.

In this chapter, we presented an empirical study upon the official IaC standard called TOSCA, for which we constructed a comprehensive dataset of publicly available blueprints, deduced the characteristics of current practices, and investigated the performance of metric- and unsupervised learning-based techniques for smell detection. The implementation is made available on Github, accompanied by an explanation for usage and research reproduction.[17]

The findings of this study are many-fold. First, we observed significat characteristical differences between smelly and sound blueprints based on their code structure for the current practices concerning TOSCA blueprint development. The findings concerning configuration smells are also noteworthy: the range of researched configuration smells in previous work is relatively small because IaC is a new research area; however, we argue for more empirical work on configuration smells to broaden the smell catalog for IaC.

Finally, researchers can enhance this work based on the constructed dataset by applying more sophisticated techniques and analysis to investigate Blob blueprints further and open opportunities for extensive studies on code smells in technology-agnostic infrastructure code.

[17]https://github.com/jade-lab/tosca-smells

7

DEFUSE: A DATA ANNOTATOR AND MODEL BUILDER FOR SOFTWARE DEFECT PREDICTION[1]

This chapter describes the final implementation of DEFUSE, which aims to help software practitioners prioritize their inspection efforts for Infrastructure-as-Code (IaC) scripts by proposing prediction models of failure-prone IaC scripts. DEFUSE implements a binary classifier either in a within-project or just-in-time set-up. More specifically, in the former, code metrics are extracted from every file at each release, and models are trained and evaluated between successive releases using those metrics. Likewise, in the latter, process metrics (i.e., metrics related to the development process such as the number of files committed together) are calculated at each commit, and models are trained and evaluated between successive commits.

7.1. ARCHITECTURE

DEFUSE is a language-agnostic framework that helps researchers collect failure-prone scripts from software repositories, check and eventually correct misclassified data, and build and evaluate machine learning models for defect prediction based on those data. Figure 7.1 shows an overview of its architecture. DEFUSE essentially consists of a dockerized application based on two modules: `defuse-ui` and `defuse-backend`. Both are Docker containers orchestrated using Docker Compose. The former is an Angular web application that guides users through repository mining, data annotation, and model building and evaluation. The latter is a Flask application that builds on top of the Python framework *RepositoryMiner* (Appendix B) to mine defect-fixing commits and failure-prone components from source code repositories. This data is used to create the dataset

[1]This was published in: **S. Dalla Palma**, D. Di Nucci, D. A. Tamburri. Defuse: a Data Annotator and Model Builder for Software Defect Prediction. *In Proceedings of the 38th IEEE International Conference on Software Maintenance and Evolution, 2022.*

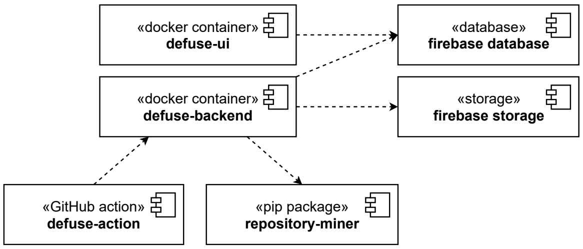


Figure 7.1: UML component diagram of DEFUSE.

that will serve as ground truth to train prediction models of failure-prone files. In addition, it provides the API endpoints used by `defuse-ui` to trigger such operations, train the predictive models, and visualize the derived data. Furthermore, both modules use the Firebase database[2] to record and access commits and model data. Moreover, the `defuse-backend` module uses Firebase storage[3] to store raw models and logs.

Finally, a GitHub Action is available for integrating DEFUSE in GitHub workflows.[4] GitHub Actions makes it easy to automate software workflows and build, test, and deploy code right from GitHub.[5] In this view, the `defuse-action` is thought to facilitate code reviews by automatically looking for defects in the committed files and stopping the workflow or notifying the user if a given defect is detected for a specified language.

7.2. USAGE TEMPLATES

The following sections illustrate how to set up and use DEFUSE.

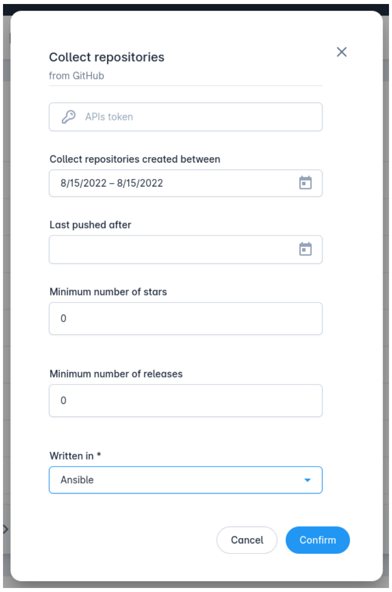
7.2.1. SETTING UP DEFUSE

For security and privacy reasons (e.g., access models of private projects), DEFUSE was designed to let users host it on their local machine or cloud providers. Therefore, the first step in setting it up is to clone the repository locally. Then, a Firebase project must be configured appropriately for data storage. A step-by-step tutorial on the GitHub repository page illustrates the complete configuration process.[6]

The web application (i.e., `defuse-ui`) is accessible on *http://localhost:4200*, while the back-end (i.e., `defuse-backend`) is on port *5000*. Nevertheless, the application can be deployed online on any hosting service supporting dockerized applications, such as Heroku. The steps to build and run DEFUSE are shown below.

```
git clone https://github.com/radon-h2020/radon-defuse
cd radon-defuse
docker-compose build
docker-compose up
```

[2]https://firebase.google.com/docs/database/
[3]https://firebase.google.com/docs/storage/
[4]https://github.com/radon-h2020/radon-defuse-action
[5]https://github.com/features/actions
[6]https://github.com/radon-h2020/radon-defuse/blob/main/README.md


Collect repositories
from GitHub

APIs token

Collect repositories created between: 8/15/2022 – 8/15/2022

Last pushed after:

Minimum number of stars: 0

Minimum number of releases: 0

Written in *: Ansible

Figure 7.2: Collecting repositories on GitHub.

7.2.2. USING DEFUSE

DEFUSE envisions four primary usage scenarios:

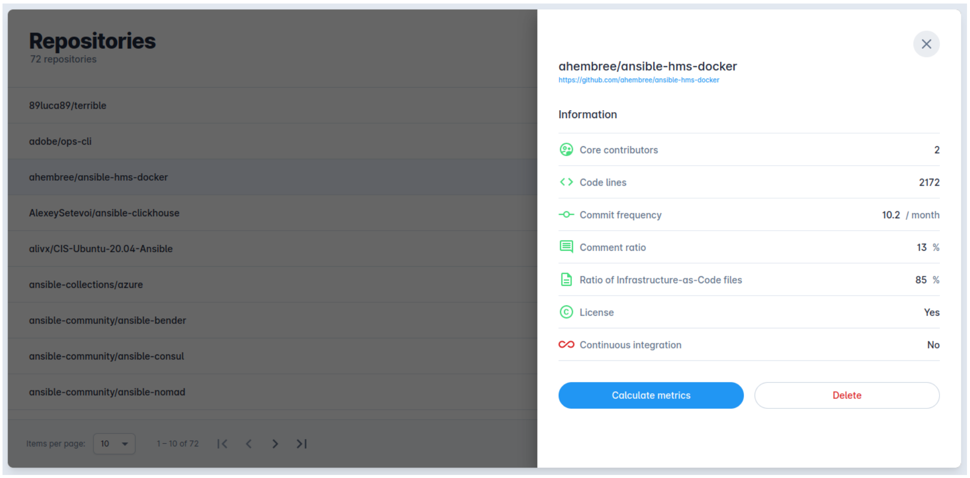
- Collect IaC-based repositories on GitHub;
- Compute repository metrics based on best engineering practices;
- Automated mining of failure-prone and neutral IaC scripts and semi-automatic commit annotation;
- Automated metrics extraction and model building.

The first step in using the tool is to connect either a public or private GitHub repository through the appropriate modal, asking for a URL (e.g., `https://github.com/ansible/ansible`) and a personal access token; the personal access token is mandatory only for private repositories and provided by GitHub. Alternatively, multiple public repositories can be collected from GitHub based on user-defined selection criteria, such as the minimum number of releases, issues, stars, and watchers, as shown in Figure 7.2. In this case, the personal access token is required for DEFUSE to use GitHub's APIs.[7]

Optionally, the user can select repositories based on metrics describing best engineering practices, as shown in Figure 7.3. Those metrics were introduced in Section 4.2 and are considered good indicators of well-engineered software projects (Munaiah et al. 2017) and include the number of core contributors, the commit, and issue frequency.[8]

[7]`https://developer.github.com/v4/`

[8]The implementation of those metrics can be found at `https://github.com/radon-h2020/radon-repository-scorer`



Repositories
72 repositories

89luca89/terrible
adobe/ops-cli
ahembree/ansible-hms-docker
AlexeySetevoi/ansible-clickhouse
alivx/CIS-Ubuntu-20.04-Ansible
ansible-collections/azure
ansible-community/ansible-bender
ansible-community/ansible-consul
ansible-community/ansible-nomad

Items per page: 10 1 – 10 of 72

ahembree/ansible-hms-docker
https://github.com/ahembree/ansible-hms-docker

Information

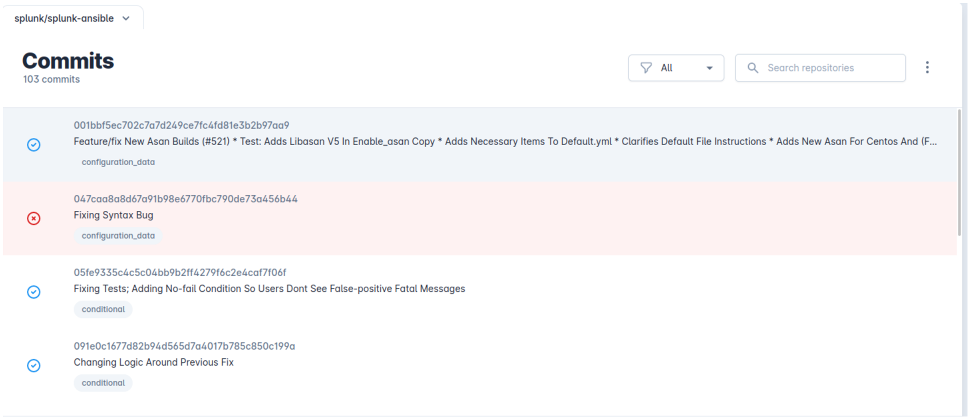
Core contributors	2
Code lines	2172
Commit frequency	10.2 / month
Comment ratio	13 %
Ratio of Infrastructure-as-Code files	85 %
License	Yes
Continuous integration	No

Figure 7.3: Repository metrics based on best engineering practices.

Afterward, the user is redirected to the dashboard, where they can mine and annotate commits of the selected repository. At this point, DEFUSE uses REPOSITORY-MINER to collect **defect-fixing commits** and **fixed files**. This process is hidden from the user, who sees the final result in two separate tables.

A defect-fixing commit in DEFUSE (shown in Figure 7.4) consists of a commit *hash*, *message*, and the *defects* it fixes based on the defect taxonomy proposed by Rahman, Farhana, Parnin, et al. 2020. Fixed files (shown in Figure 7.5) are files modified in defect-fixing commits and consist of a *fixing commit*, the hash of the *bug-introducing commit*, and the *path* of the file deemed to be fixing the bug. Of course, not all files modified in a defect-fixing commit contribute to the fix. For example, some files may have been just refactored. Therefore, these files should not be used for training a defect-prediction model. However, DEFUSE assumes that the files modified by a defect-fixing commit solely fix a bug. Hence, it lets the user discard files categorized as fixing a bug mistakenly, a.k.a. false positives. False positives can be filtered manually by clicking on the round checkbox dots in the tables (which transit to red). The related instances will be ignored during model training. More specifically, excluding a commit excludes all the files modified in that commit.

Finally, a model can be trained for a specific defect and language in the Model Manager dashboard (Figure 7.6). At this point, DEFUSE uses *RepositoryMiner* to label as failure-prone all the snapshots of a fixed file in the releases within its bug-introducing and defect-fixing commit. Conversely, snapshots of "non-fixed files" are labeled as neutral. Then, code metrics are calculated for every snapshot to create the dataset used as ground truth for training the defect prediction model. Finally, the resulting model can be downloaded for custom usage, checked for performance (Figure 7.7), or used for CI/CD using the provided GitHub Action.


splunk/splunk-ansible

Commits
103 commits

All | Search repositories

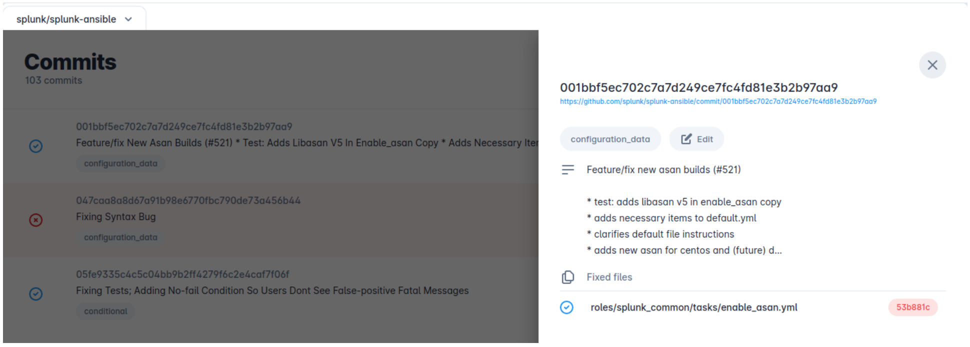
001bbf5ec702c7a7d249ce7fc4fd81e3b2b97aa9
Feature/fix New Asan Builds (#521) * Test: Adds Libasan V5 In Enable_asan Copy * Adds Necessary Items To Default.yml * Clarifies Default File Instructions * Adds New Asan For Centos And (F...
configuration_data

047caa8a8d67a91b98e6770fbc790de73a456b44
Fixing Syntax Bug
configuration_data

05fe9335c4c5c04bb9b2ff4279f6c2e4caf7f06f
Fixing Tests; Adding No-fail Condition So Users Dont See False-positive Fatal Messages
conditional

091e0c1677d82b94d565d7a4017b785c850c199a
Changing Logic Around Previous Fix
conditional

Figure 7.4: Table of defect-fixing commits. Files modified in red-highlighted commits are ignored from training.


splunk/splunk-ansible

Commits
103 commits

001bbf5ec702c7a7d249ce7fc4fd81e3b2b97aa9
Feature/fix New Asan Builds (#521) * Test: Adds Libasan V5 In Enable_asan Copy * Adds Necessary Iter
configuration_data

047caa8a8d67a91b98e6770fbc790de73a456b44
Fixing Syntax Bug
configuration_data

05fe9335c4c5c04bb9b2ff4279f6c2e4caf7f06f
Fixing Tests; Adding No-fail Condition So Users Dont See False-positive Fatal Messages
conditional

001bbf5ec702c7a7d249ce7fc4fd81e3b2b97aa9
https://github.com/splunk/splunk-ansible/commit/001bbf5ec702c7a7d249ce7fc4fd81e3b2b97aa9

configuration_data | Edit

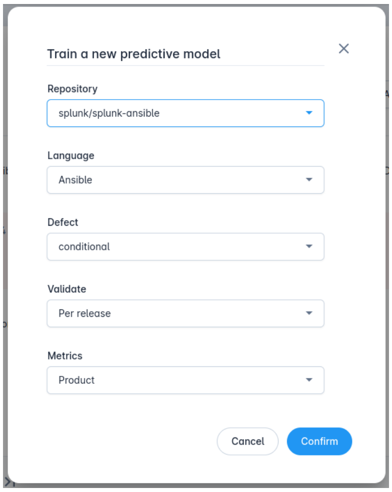
Feature/fix new asan builds (#521)

* test: adds libasan v5 in enable_asan copy
* adds necessary items to default.yml
* clarifies default file instructions
* adds new asan for centos and (future) d...

Fixed files

roles/splunk_common/tasks/enable_asan.yml 53b881c

Figure 7.5: Table of fixed files.



Train a new predictive model

Repository: splunk/splunk-ansible

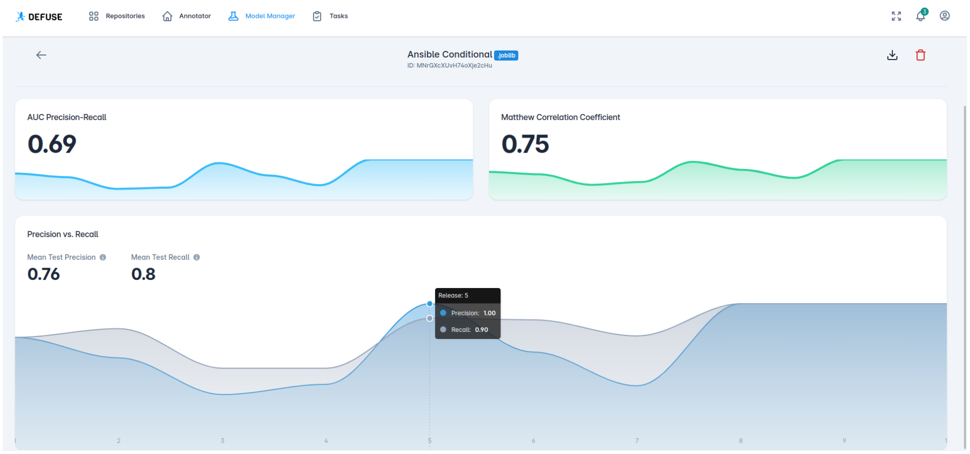
Language: Ansible

Defect: conditional

Validate: Per release

Metrics: Product

Figure 7.6: Training a defect prediction model.



Ansible Conditional .joblib
ID: MNrGXcXUvH74oXje2cHu

AUC Precision-Recall 0.69

Matthew Correlation Coefficient 0.75

Precision vs. Recall

Mean Test Precision 0.76

Mean Test Recall 0.8

Release: 5
Precision: 1.00
Recall: 0.90

Figure 7.7: Model evaluation GUI.

Listing 10 A generic GitHub workflow using the Action.

```
on: [push]

jobs:
  defect_prediction_job:
    runs-on: ubuntu-latest
    name: Run Defuse model M for language L
    steps:
      - name: Defect Prediction
        id: defect_prediction
        uses: radon-h2020/radon-defuse-action@v1.1
        with:
          model: <model id>
          language: <ansible|tosca>
          url: <URL to defuse-backend>
```

7.2.3. INTEGRATION INTO GITHUB WORKFLOWS

With GitHub Action, user interaction with the model is minimal. Listing 10 shows that the user needs to specify only three parameters to create and run the GitHub workflow:

- The model *id*; it can be found on the model manager page.
- The *language* of the files to scan; currently, “ansible” or “tosca”.
- The *URL* to `defuse-backend` in the deployed application.

7.2.4. INTEGRATION INTO CUSTOM WORKFLOWS

While GitHub Action eases the interaction with the APIs, it also limits the usage within GitHub workflows. If the user wants to use a trained model in their workflow (e.g., in a Jenkins instance or GitLab workflow), that is possible by calling the *predict/* endpoint available at the *defuse-backend* URL. It enables calls to a trained model for predictions from either the application itself or third-party applications. It requires a model id – i.e., `model_id` – and a list of key-value pairs representing metric names and their value – e.g., `lines_code=10&num_tasks=3`. For example, Listing 11 shows a Python script that extracts Ansible source code metrics using the library *ansiblemetrics* (Dalla Palma, Di Nucci, and Tamburri 2020) and creates the URL for the prediction by concatenating the metrics' name and value in the query and sends the request to the APIs. Here, the model with id `model_id` is supposed to be trained on metrics extracted using the same library. The response is a JSON object containing the prediction (i.e., failure-prone or neutral) and its explanation.

7.3. LIMITATIONS AND EXTENSIONS

This section illustrates some of the limitations of the tool and possible extensions. First, DEFUSE is currently limited to two IaC languages, namely Ansible and Tosca, but we plan to support more common languages in the future, such as Java and Python.

As for the classifiers, DEFUSE uses a Decision Tree to build the prediction model to build the prediction model because of its interpretability and performance, as observed in Chapter 5. However, we plan to add a GUI component to select the machine learning

Listing 11 A Python template to invoke the prediction.

```python
import requests
from ansiblemetrics import metrics_extractor

url = f'<url/to/defuse-backend>/predict?model_id=<model id>'

metrics = metrics_extractor.extract_all('Paste a valid YAML here')

for name, value in metrics.items():
    url += f'&{name}={value}'

response = requests.get(url)
print(response.content)

# Example of output:
# { "failure-prone": False,
#   "decision": [["lines_code", "<=', 100], ["num_tasks", "<=", 5]]}
```

technique for training. Furthermore, the tool detects defects at the file level. Future implementation will target lower levels of granularity, such as features, methods, and code lines.

In addition, DEFUSE uses the SZZ implemented in PyDriller. Although we observed high precision in the scope of IaC, we plan to evaluate and eventually add the Refactoring-Aware SZZ (Neto et al. 2018) as it can reduce false positives caused by refactoring.

Then, as one of the most critical aspects of building a model, we plan to empower the current annotation with active learning. The key idea behind active learning is that a machine learning algorithm can achieve greater accuracy with fewer labeled training instances if it can choose the data from which it learns (Settles 2009). Since the user chooses the examples, the number of examples to learn a concept can often be much lower than the number required in supervised learning.

Finally, the end-user must stick to the Firebase billing plans because of the dependency on this platform. DEFUSE works with the free Firebase Spark plan, which offers generous limits for getting started with Firebase; however, users can upgrade to the paid-tier Blaze plan without affecting the tool's functioning.[9]

7.4. SUMMARY

This chapter presented the final implementation of DEFUSE, the language-agnostic tool for software defect prediction to automatically collect and classify failure data and build machine learning models to detect defects based on those data.

The key characteristics of the tool are (i) a commit annotator of defect-fixing commits and failure-prone components; (ii) a model builder for defect prediction; and (iii) a GitHub action to integrate the tool in GitHub workflows.

We deem DEFUSE will strive to save costs by instrumenting more focused maintenance and testing and avoiding erroneous execution risks that can lead to hazardous or even harmful infrastructure or execution failures.

[9] https://firebase.google.com/docs/projects/billing/firebase-pricing-plans

8

Conclusions, Lessons Learned, and Open Issues

This dissertation had the following research objective:

To advance the state-of-the-art and state-of-the-practice of Infrastructure-as-Code (IaC) quality assurance, focusing on developing tools for code smell detection and defect prediction for IaC using Machine Learning and their evaluation by large-scale empirical studies.

We put forward the following two research questions along with several sub-questions to accomplish the objective:

RQ_1 What code metrics can be employed to characterize the quality of, and identify problems in, Infrastructure-as-Code?

RQ_2 Can a Machine Learning approach be employed during IaC Quality Assurance to accurately detect defects and code smells using the elicited metrics?

- $RQ_{2.1}$ How is the prediction performance affected by choice of the classifier selection?
- $RQ_{2.2}$ How is the prediction performance affected by choice of the metric sets?
- $RQ_{2.3}$ What metrics are the most effective in maximizing defect prediction performance?
- $RQ_{2.4}$ How can structural code metrics distinguish between Blob – i.e., too large and complex – and sound blueprints?
- $RQ_{2.5}$ To what extent can metric- and unsupervised learning-based techniques detect those lousy blueprints?
- $RQ_{2.6}$ What metrics are the most effective in maximizing the performance of those detectors?

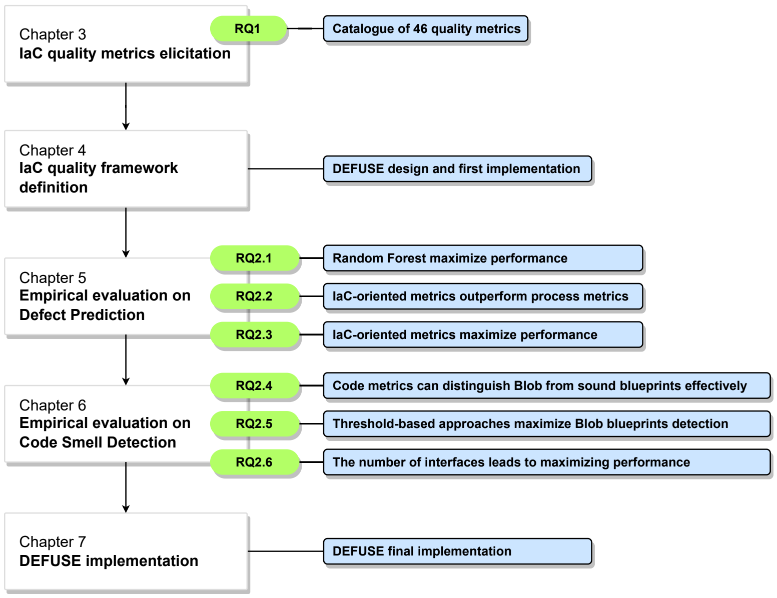

Chapter 3
IaC quality metrics elicitation
RQ1
Catalogue of 46 quality metrics
Chapter 4
IaC quality framework definition
DEFUSE design and first implementation
Chapter 5
Empirical evaluation on Defect Prediction
RQ2.1
Random Forest maximize performance
RQ2.2
IaC-oriented metrics outperform process metrics
RQ2.3
IaC-oriented metrics maximize performance
Chapter 6
Empirical evaluation on Code Smell Detection
RQ2.4
Code metrics can distinguish Blob from sound blueprints effectively
RQ2.5
Threshold-based approaches maximize Blob blueprints detection
RQ2.6
The number of interfaces leads to maximizing performance
Chapter 7
DEFUSE implementation
DEFUSE final implementation

Figure 8.1: Mapping between research questions and contributions. While Chapter 3, 5, and 6 addresses the research questions, Chapter 4 and 7 are prototypical design chapters that aim at describing the framework and its implementation.

8.1. CONCLUSIONS & REFLECTIONS

The answers to the high-level research questions are summarized as follows.

RQ_1 WHAT CODE METRICS CAN BE CAN BE EMPLOYED TO CHARACTERIZE THE QUALITY OF, AND IDENTIFY PROBLEMS IN, INFRASTRUCTURE-AS-CODE?

For research question RQ_1, we explored existing literature on application and infrastructure code to devise IaC quality metrics. We proposed a catalog of 46 metrics that can be used to evaluate the different aspects of IaC – the most comprehensive measures set for IaC to date. The catalog consists of:

- Eight metrics relate to source code characteristics, language-agnostic.
- Fourteen metrics from existing works on IaC research but for different languages.
- Twenty-four metrics concerning best and bad practices in Ansible, observable in different orchestration configuration languages.

The conjecture behind the proposed metrics is that, generally, high values make the infrastructure code complex and challenging to interpret, debug, and maintain.

Furthermore, the metrics can be categorized in terms of their scope (i.e., general, playbook, and tasks list) depending on the particular construct or artifact they target. General metrics can apply to either playbooks and task files, i.e., files that contain a flat list of tasks, but can also generalize to other languages. *Playbook* metrics operate within a single playbook, while *tasks list* metrics can operate within a playbook and tasks. Although their implementation targets Ansible, they are equivalent (and portable) to other languages (e.g., Chef, Puppet) and offer a general-purpose metrics-based approach for IaC quality evaluation.

RQ_2 Can a Machine Learning approach be employed during IaC Quality Assurance to accurately detect defects and code smells using the elicited metrics?

For research question RQ_2, specifically $RQ_{2.1}$ – $RQ_{2.3}$, in Chapter 5, we showed how the proposed infrastructure source code properties could be used and combined as surrogate metrics for defect-proneness of infrastructure components and to identify defects in Ansible. We experimented with those metrics, along with process metrics and groups thereof, in the context of within-project defect prediction to evaluate their ability to distinguish failure-prone scripts from neutral scripts and predict their failure-proneness.

$RQ_{2.1}$ How is the prediction performance affected by the choice of the classifier selection? Concerning $RQ_{2.1}$, we observed that models trained using Random Forest perform statistically better than those relying on the remaining classifiers. The difference is statistically different with a large effect size. Nevertheless, we observed that the performance difference between Support Vector Machine and Logistic Regression and between Logistic Regression and Decision Tree is not statistically significant, although the former provided better results most of the time. At the same time, they have good performance, although lower than Random Forest. Consequently, depending on the desired model flexibility and the available computational resources, one can choose them interchangeably without negatively affecting the prediction. We used the Decision Tree classifier in the final implementation for model and prediction interpretability (Chapter 7).

$RQ_{2.2-2.3}$ How is the prediction performance affected by the choice of the metric sets? and What metrics are the most effective to maximize defect prediction performance? In $RQ_{2.2}$ and $RQ_{2.3}$, we found that IaC-oriented product metrics are good indicators of failure-prone IaC scripts and that models trained on those metrics outperform those trained on process metrics or a combination thereof. Strengthening this result, we observed that the top 13 predictors – in terms of occurrences among the most critical features resulting from a recursive feature elimination procedure – include IaC-oriented metrics only. Therefore, we conclude that structural code metrics outperform process metrics for the collected Ansible-based projects, 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 changes to infrastructure code compared with changes to application code, thus, limiting the information exploitable by the process metrics. Consequently, we used only product metrics in the final tool implementation (Chapter 7).

$RQ_{2.4}$ To what extent can structural code metrics distinguish between Blob – i.e., too large and complex – and sound blueprints? Concerning IaC smell detection, in Chapter 6, we enhanced the current knowledge of current practices in Infrastructure-as-Code, and the detection of configuration smells. As indicated by Rahman, Mahdavi-Hezaveh, et al. 2019, characteristics of best practices within IaC are insufficiently addressed in current scientific work, and only a handful of previous works investigated configuration smells. Therefore, we conducted a study on the official OASIS standard for IaC, TOSCA; we constructed a comprehensive dataset of publicly available blueprints, deduced the characteristics of current practices, and investigated the performance of metric- and unsupervised learning-based techniques for smell detection. For the current practices concerning TOSCA blueprint development, we observed significant differences between smelly and sound blueprints based on their code characteristics, such as the number of code lines, the number of topology types and templates, and the number of interfaces and properties, and cohesion.

$RQ_{2.5}$ TO WHAT EXTENT CAN METRIC- AND UNSUPERVISED LEARNING-BASED TECHNIQUES DETECT THOSE LOUSY BLUEPRINTS? In research question $RQ_{2.5}$, we showed that the metrics resulting from $RQ_{2.4}$ could be used as accurate early detectors of Blob blueprints in both metric-based and unsupervised-learning-based detectors, although the former generally perform better.

$RQ_{2.6}$ WHAT METRICS ARE THE MOST EFFECTIVE IN MAXIMIZING THE PERFORMANCE OF THOSE DETECTORS? Finally, in research question $RQ_{2.6}$, we observed that, among the considered metrics, the number of interfaces appears to be a leading metric to maximize the detection of Blob blueprints; the number of types and templates, code lines, and lack of cohesion follows.

8.2. LESSONS LEARNED

The main lessons learned from this thesis can be summarized as follow.

USE MODEL INTERCHANGEABLY WITHOUT SIGNIFICANTLY NEGATIVELY AFFECTING THE PREDICTION

Regardless of the metrics used, Random Forest over-performs other learning techniques when predicting failure-prone IaC scripts. Nevertheless, we observed high performance in the remaining techniques. Among them, the performance difference between Support Vector Machine and Logistic Regression and between Logistic Regression and Decision Tree is not statistically significant, although the former provided better results most of the time. Consequently, depending on the desired model flexibility and the available computational resources, one can choose them interchangeably without negatively affecting the prediction.

PRODUCT METRICS OUTPERFORM PROCESS METRICS

For the collected Ansible-based projects, structural code metrics outperform process metrics, although the latter is often more effective when predicting the failure-proneness of source code instances in traditional defect prediction. Not only, but IaC-oriented metrics maximize the prediction performance most of the time. In particular, the numbers of tokens and code lines in the blueprint and its text entropy are the most occurring predictors. Practitioners that 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 one mentioned above. This aspect will reduce the number of features to collect, reduce the number of tools required to mine them, and speed up the training phase.

Refactor Nodes Based on Target Technology

While validating Blob blueprints, in Chapter 6, we observed several defining too many nodes and targeting different technologies. Based on that observation, we advise refactoring those types in different blueprints to reduce the complexity, each grouping type targeting the same or similar technology. Then, those blueprints can be imported into the one at hand.

Refactor Nodes Based on Types

TOSCA provides types describing the possible building blocks for constructing a service template. For example, node types describe the kinds of nodes, relationship types describe possible relations among those nodes, and policy types logically group related nodes that need to be orchestrated or managed together to achieve results. While it is possible and might be advisable for a blueprint to define one or more components of each type, too many different types hinder the blueprint's clarity. Instead, those blueprints could be refactored into separate files for each type or group.

Move Workflows Into Their Separate Files

Blueprints can have long workflows that significantly increase the size and decrease readability – workflows are used for topology (un)deployment or run-time management. In those cases, blueprints could benefit if workflows were defined in their files and imported into the blueprint.

8.3. Open Issues

Despite the effort devoted by the research community and despite the advances proposed in this thesis, Infrastructure-as-Code maintenance still proposes some open issues and challenges related to quality aspects that need to be addressed in the future. Given the early-stage research on IaC analytics, there are several avenues for future work.

Relationship of Metrics and Code Quality

We know that some of the proposed metrics may have little effect on code quality. However, future empirical studies are required to accurately identify and validate what metrics significantly affect a given quality aspect of a configuration management system. First, more research is needed to understand the relation between the presented metrics and the quality of IaC blueprints at different granularity levels, e.g., using those metrics to measure the quality of an Ansible role – i.e., re-usable Ansible components – rather than a playbook or a specific task. Second, there is still a lack of empirical evidence regarding the scope and usability of such measures, e.g., whether they can be used or combined to detect code smells and bugs. This thesis has already stepped up in this

```yaml
- name: Install ansiblemetrics
  pip:
    name: ansiblemetrics
    state: latest


- name: Retrieve file status for foo.yaml
  stat:
    path: /playbooks/foo.yaml
  register: foo

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

```hcl
resource "null_resource" "ansiblemetrics" {
  provisioner "local-exec" {
    command = "pip install ansiblemetrics"
  }
}

data "file" "foo" {
  filename = "/playbooks/foo.yaml"
}

resource "null_resource" "run_ansiblemetrics" {
  depends_on = [null_resource.ansiblemetrics]

  provisioner "local-exec" {
    command = "ansible-metrics
    ↪  /playbooks/foo.yaml --dest report.json"
    working_dir = "/playbooks"
    when = data.file.foo.exists
  }
}
```

Listing 12: Ansible tasks (left) and Terraform resources (right) to install ANSIBLEMETRICS from PyPI and run it.

direction by evaluating the proposed metrics as a proxy of defect prediction models for IaC scripts. A further step in this direction is mapping the proposed metrics to the software quality attributes they model, such as maintainability, complexity, re-usability, and more.

METRICS GENERALIZATION AND CROSS-PROJECT DEFECT PREDICTION

Further research is needed to understand the relationship between the failure-prone Infrastructure-as-Code and the collected metrics. There is still room for further research in this area. Nevertheless, 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., based on the number of core contributors, number of commits, lines of code, and the ratio of IaC files. This aspect is particularly interesting for cross-project defect prediction, where the lack of historical data forces organizations to use pre-trained models built on similar projects. These results can lead to a better understanding of which features to utilize to improve defect prediction of IaC.

Besides, the catalog proposed in Chapter 3 can be further generalized to additional configuration management and orchestration languages such as Chef, Puppet, and Terraform. More specifically, mapping Ansible-specific characteristics to other languages involves identifying and matching the characteristics of Ansible with the equivalent constructs in other languages. For example, Terraform is another tool for automating infrastructure that uses a declarative, high-level configuration language to define and provision a wide range of cloud and on-premises resources in a reproducible and scalable manner. While it has different approaches and constructs from Ansible, there are some similarities that can facilitate mapping between them. For instance, Ansible modules can be mapped to Terraform resources, which are the fundamental unit for representing infrastructure components and services,[1] and Terraform variables can be mapped

[1] https://developer.hashicorp.com/terraform/language/resources/syntax

to Ansible variables. Similarly, Terraform modules can be mapped to Ansible roles, and Ansible lookup can be mapped to Terraform data sources as both retrieve external data that can be used within the configuration, although they have different approaches and syntax. These mappings can be useful for leveraging existing knowledge and tools for Ansible to work with other languages.

An example of Ansible-Terraform mapping is shown in Listing 12, where both Ansible and Terraform scripts ensure the installation of the latest version of *ansiblemetrics* and run the tool on a file called *foo.yml* present in the folder *playbooks*. Besides text-based metrics that can be easily extracted from both scripts, such as the number of code lines and comments, text entropy, and the number of tokens, metrics such as the number of modules and file existence checks can be applied to both Ansible and Terraform scripts. Ansible modules can be translated into Terraform resources, and the Terraform data block can be used to retrieve data from an external source.

Overall, the metrics proposed for Ansible in this thesis can also be applicable to other languages – an example of mapping to Puppet is provided in Appendix A. Researchers can use these mappings to ensure that the infrastructure code follows best practices and high-quality standards.

NEEDS FOR EMPIRICAL STUDIES ON IAC SMELL DETECTION

Although many code smells were proposed and studied for traditional programs, IaC-specific smells and, in particular, their detection techniques remain unripe. In addition, characteristics of best practices within IaC are insufficiently addressed in current scientific work, and the range of researched configuration smells in previous work is relatively small because IaC is a new research area.

Although our work in Chapter 6 extends the current knowledge regarding the development behavior for technology-agnostic infrastructure code by investigating how this behavior relates to large and complex blueprints, we argue for more empirical work on configuration smells to broaden the smell catalog for IaC. Finally, other researchers can enhance this work based on the constructed dataset by applying more sophisticated techniques and analysis to investigate Blob blueprints further and open opportunities for extensive studies on code smells in technology-agnostic infrastructure code.

8.4. SUMMARY

This chapter summarises the main findings of the studies reported in this dissertation, the lesson learned, and open issues and challenges related to quality aspects that need to be addressed in the future. We also illustrated the final architecture of the tool and its evaluation, and provided examples of its programming interface. Given the early-stage research on IaC analytics, there is still room for improvements: researchers can build on these results and use the shared material as a baseline to better understand failure-prone infrastructure code and compare competing approaches for defect prediction. Furthermore, researchers gain valuable knowledge of novel IaC languages and tools, comprehensive datasets and metrics sets, and predictive models. In short, this research puts a clean baseline for existing and future approaches.

Finally, we received encouraging feedback from the industrial partners within the

European Union's project RADON (https://radon-h2020.eu). Therefore, we hope and plan to enlarge the user community around DEFUSE through their collaboration and dissemination. We are confident that the proposed tool will help researchers and practitioners collect meaningful data to feed defect prediction models for infrastructure and application code and benefit from a common tool for defect prediction.

APPENDIX

A
AnsibleMetrics: A Python Library for Measuring Infrastructure-as-Code Blueprints in Ansible[1]

Despite the growing interest in Infrastructure-as-Code (IaC), only a few works and tools dealt with the quality of IaC. For example, Bent et al. 2018 developed a measurement model and a tool to analyze the Puppet code's maintainability. They showed that the measurement model provides quality judgments of Puppet code that closely match the experts' recommendations through structured interviews with experts. Other works focused on the analysis of source code properties to build machine-learning models that evaluate the defectiveness of IaC scripts (Rahman and Williams 2018) or the occurrence of security smells in infrastructure code, such as the Security Linter for Infrastructure-as-Code scripts (Rahman, Parnin, et al. 2019), initially developed for Puppet. Finally, Ansible Lint[2] helps developers deal with infrastructure code by checking playbooks for practices and behavior that could be improved. These include formatting and idiom issues, such as trailing whitespace and comparison to an empty string, respectively; tasks issues such as unnamed tasks, and more.

This chapter presents one of the tools we developed to build a more robust and variegate suite on this line of research: AnsibleMetrics, a Python-based static source code analyzer for Ansible blueprints that helps quantify the characteristics of Infrastructure-as-Code to support DevOps engineers when maintaining and evolving it. It currently supports the 46 source code metrics proposed in Chapter 3 that include the number and the size of plays and tasks, the number of commands, and best and bad practices, though

[1]This chapter was published in: **S. Dalla Palma**, D. Di Nucci, and D. A. Tamburri. AnsibleMetrics: A Python Library for Measuring Infrastructure-as-Code Blueprints in Ansible. *SoftwareX, 2020.*

[2]https://github.com/ansible/ansible-lint (Accessed September 2020)

others can be derived by combining the implemented ones. The metrics were extracted from the Ansible documentation and the existing scientific literature on object-oriented metrics and IaC measures from Puppet. Please, see the documentation[3] for further details about each metric.

Although some metrics implemented in ANSIBLEMETRICS might overlap with the checks defined in ANSIBLE LINT, the tools have different goals. The latter focuses on best practices, while the former is intended to identify source code properties and statistics that can be used as early indicators of faulty infrastructure scripts, potentially leading to expensive infrastructure failure[4]. Therefore, we deem the two tools could be used in conjunction. Indeed, ANSIBLE LINT identifies violations of best practices defined by rules. At the same time, the metrics extracted in ANSIBLEMETRICS could be combined to characterize failure-prone IaC scripts in Software Defect Prediction (Hall et al. 2012) or to identify symptoms that indicate wrong style usage or a lousy design, a.k.a. code smells (Fowler 2018).

The tool is open-source on GitHub[5] and available as a Python package[6] along with documentation.

The remainder of this chapter is structured as follows. Appendix A.1 describes the main APIs; Appendix A.2 describes the tool's architecture; Appendix A.3 provides illustrative examples to understand the characteristics and use of the proposed tool.

A.1. DOMAIN OBJECTS

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

- `AnsibleMetric` is an abstract class extended by all the classes responsible for computing metrics for a given Ansible construct, such as the number of plays, tasks, and modules. It takes a plain YAML file as input and stores it as a list of dictionaries (note that the dictionary is the datatype provided by the library *PyYAML* used to parse it). In addition, it extracts the plays and tasks from the input script and stores them as a list of dictionaries in the class attributes `plays` and `tasks`, respectively. This step facilitates the metrics implementation: no further check concerning the structure of plays and tasks is needed.

- `LinesMetric` is an abstract class responsible for computing language-agnostic metrics related to code lines, such as the number of executable lines, blank lines, and comments. Similar to `AnsibleMetrics`, it takes a plain YAML file as input. That file is stored as a string to allow for parsing comments and blank lines, which `PyYAML` ignores.

Both classes validate the YAML file in their *__init__()* method and raise a *ValueError* whenever the file is empty or is not well-formatted. Therefore, the extended classes

[3]https://radon-h2020.github.io/radon-ansible-metrics/
[4]https://cloudcomputing-news.net/news/2017/oct/30/glitch-economy-counting-cost-software-failures/ (Accessed September 2020)
[5]https://radon-h2020.github.io/radon-ansible-metrics/
[6]https://pypi.org/project/ansiblemetrics/

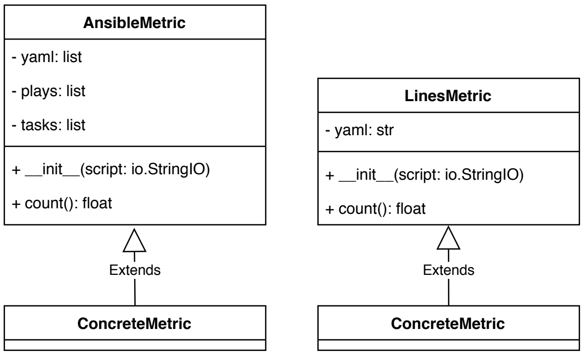
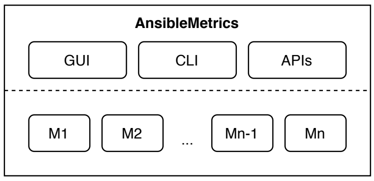
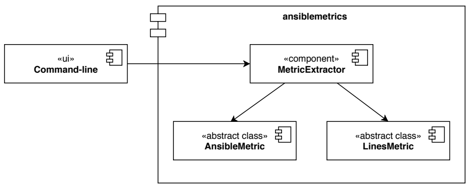


Figure A.1: Diagram of the two base classes of ANSIBLEMETRICS.



(a) A logical schema of the tool.



(b) A component diagram of the ANSIBLEMETRICS architecture.

Figure A.2: Architecture overview.

(`ConcreteMetric` in Figure A.1) can safely assume the file is a valid YAML when parsing it and implementing the logic of the method *count()*.

A.2. SOFTWARE ARCHITECTURE

ANSIBLEMETRICS presents a two-level software architecture as schematized in Figure A.2a. The higher level is composed of the user interfaces (GUI and Command-line) and code APIs. They provide the main entry points for third-party applications that need to quantify their infrastructure code characteristics.

The GUI[7] consists of a Visual Studio Code (VSC) extension available on the VSC Marketplace[8]. The CLI takes the path of an Ansible blueprint or a directory of blueprints; optionally, it saves the report in JSON format to the path specified by the user. The JSON report consists of an object containing *(key, value)* pairs, where the *key* is the metric name and the *value* is the computed value, similar to the following:

```
{
    "avg_task_size": <integer>,
    "num_tasks": <integer>,
    "num_plays": <integer>,
```

[7]Source available on GitHub at https://github.com/radon-h2020/radon-ansible-metrics-plugin
[8]https://marketplace.visualstudio.com/items?itemName=radon-h2020.ansiblemetrics

Listing 13 An example of Ansible playbook.

```yaml
---
- name: Update webserver
  hosts: webservers
  vars:
    http_port: 80
  remote_user: root

  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest

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

  tasks:
  - name: ensure postgresql is at the latest version
    yum:
      name: postgresql
      state: latest

  - name: ensure that postgresql is started
    service:
      name: postgresql
      state: started
```

```
      "text_entropy": <float>
  }
```

The APIs allow for two primary usage scenarios:

1. `MetricExtractor` (Figure A.2b) programmatically runs all the implemented metrics. It returns a JSON object parsable by the developer, as for the command-line interface. The CLI relies on `MetricExtractor`.

2. In turn, `MetricExtractor` relies on the classes that extend `AnsibleMetric` and `LinesMetric`. A DevOps engineer can programmatically run a metric depending on the problem without extracting all the metrics simultaneously. In that case, the value returned by the metric is the one returned by its method *count()*, that is, an integer or floating number.

A.3. Usage Examples

For illustrative purposes, let us consider Listing 13. To analyze this blueprint, the user has to install the `ansiblemetrics` library from PyPI with the command:

```
pip install ansiblemetrics
```

Alternatively, from the source code root folder:

```
pip install .
```

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

A.3.1. COMMAND-LINE USAGE

Assuming that the example in Listing 13 is named *playbook1.yml* and is located within the folder *playbooks* as follows:

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

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

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

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

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

For simplicity, we reported only the non-zero metrics in this example, while the report contains all metrics by default. To omit the zero metrics in the report, the user has to pass the parameter `-omit-zero-metrics`.
The user can input the path to the folder containing them to avoid running the command multiple times for multiple blueprints. For example:

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

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

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

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

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

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

A.3.2. PYTHON MODULE USAGE

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

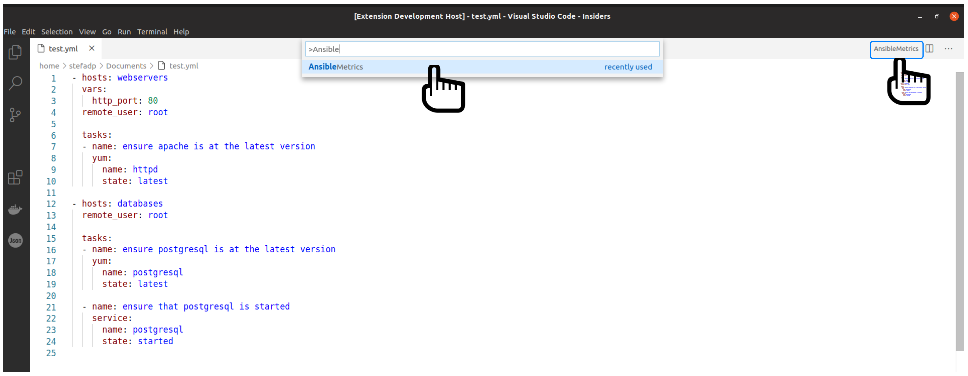
Listing 14 exemplifies these scenarios. In that code snippet, `all_metrics` will store a JSON object, as for the example in Appendix A.3.1, while the `loc_count` and `plays_count` will contain the values 20 and 2, respectively.

A.3.3. GUI USAGE

Figure A.3 and Figure A.4 depict two parts of the Graphical User Interface. The former shows how to run the plugin, while the second is an example of the report. The GUI facilitates the tool usage through editors that support the Visual Studio Code (VS Code) extension. First, the user has to install the `ansiblemetrics` dependency, as described

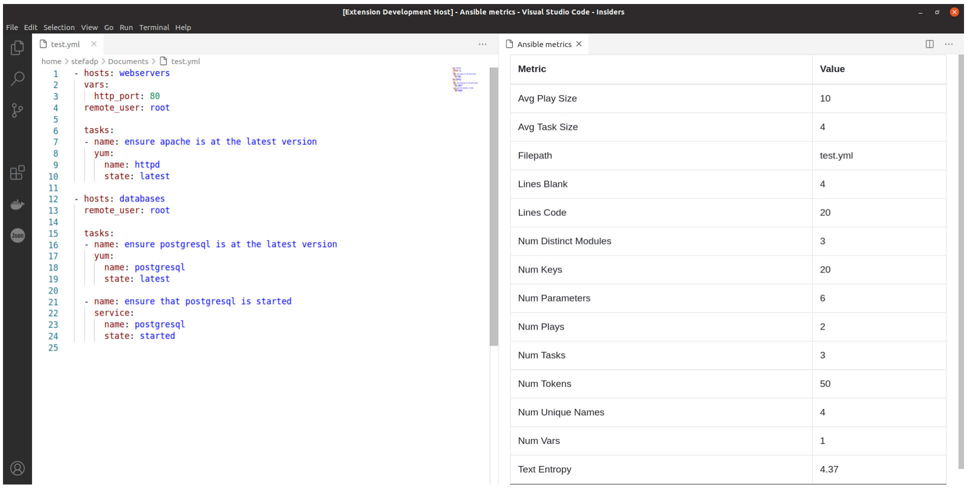
Listing 14 Example of Python usage of ANSIBLEMETRICS.

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

script = "<to be filled with example in Listing 12>"

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

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


[Extension Development Host] - test.yml - Visual Studio Code - Insiders
File Edit Selection View Go Run Terminal Help
test.yml
home > stefadp > Documents > test.yml
>Ansible
AnsibleMetrics
recently used
AnsibleMetrics

```yaml
- hosts: webservers
  vars:
    http_port: 80
  remote_user: root

  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest

- hosts: databases
  remote_user: root

  tasks:
  - name: ensure postgresql is at the latest version
    yum:
      name: postgresql
      state: latest

  - name: ensure that postgresql is started
    service:
      name: postgresql
      state: started
```

Figure A.3: How to run the plugin.


[Extension Development Host] - Ansible metrics - Visual Studio Code - Insiders
File Edit Selection View Go Run Terminal Help
test.yml
home > stefadp > Documents > test.yml
Ansible metrics

```yaml
- hosts: webservers
  vars:
    http_port: 80
  remote_user: root

  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest

- hosts: databases
  remote_user: root

  tasks:
  - name: ensure postgresql is at the latest version
    yum:
      name: postgresql
      state: latest

  - name: ensure that postgresql is started
    service:
      name: postgresql
      state: started
```

Metric	Value
Avg Play Size	10
Avg Task Size	4
Filepath	test.yml
Lines Blank	4
Lines Code	20
Num Distinct Modules	3
Num Keys	20
Num Parameters	6
Num Plays	2
Num Tasks	3
Num Tokens	50
Num Unique Names	4
Num Vars	1
Text Entropy	4.37

Figure A.4: Example of report.

```
1   - name: Ensure ansiblemetrics is at its latest version
2     pip:
3       name: ansiblemetrics
4       state: latest
5
6   # Ensure that foo.yaml exist
7   - name: Retrieving file status
8     stat:
9       path: /playbooks/foo.yaml
10    register: foo
11
12  - name: Run ansible-metrics if foo.yaml exists
13    shell:
14      chdir: playbooks/
15      cmd: 'ansible-metrics foo.yml --dest report.json'
16    when: foo.stat.exists is defined and foo.stat.exists
```

```
1   # Ensure ansiblemetrics is at the latest version
2   packages {'ansiblemetrics':
3     ensure   => 'latest',
4     provider => 'pip',
5   }
6
7   # Ensure that foo.yaml exist
8   file {'foo.yaml':
9     ensure => 'file',
10    path   => '/playbooks/foo.yaml',
11  }
12
13  exec {'run ansible-metrics':
14    cwd     => 'playbooks/',
15    command => 'ansible-metrics foo.yml --dest report.json',
16  }
```

Figure A.5: Ansible tasks (left) and Puppet resources (right) to install AnsibleMetrics from PyPIP and run it.

in the previous section. It is then possible to install the extension by directly looking for it on the VS Marketplace[9] or launching the VS Code Quick Open (`Ctrl+P`), paste the following command, and press enter:

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

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

A.4. Extending AnsibleMetrics

Although AnsibleMetrics focuses on Ansible, most metrics are easily extendible to other languages, such as Puppet. One example is listed in Figure A.5. The two code snippets are analogous: they ensure that the latest version of `ansiblemetrics` is installed, and run the tool via command-line on a file called *foo.yml* present in the folder *playbooks*. In this example, text-based metrics, such as *line of code and comments*, *text entropy*, and the *number of tokens*, can be easily extracted from both scripts; other metrics can be easily adapted. For example, the Ansible code in Figure A.5 (left) has *three* tasks (lines 1, 7, and 12), each of them calling a module[10] (*pip*, *stat*, and *shell*, respectively). Here, the metric *number of modules = 3*. *Modules* are fundamental in Ansible as they provide units of code directly executable on remote hosts or through playbooks.

Similarly, the Puppet code in Figure A.5 (right) consists of *three* resources[11] (lines 2, 8, and 13). Puppet *resources* are the fundamental unit for modeling system configurations. A Puppet resource can be considered analogous to an Ansible module. For example, the metric *number of modules = 3* can be translated into *numbers of resources = 3*. More specifically, it can be generalized to other languages, for example, as the *number of fundamental units*.

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

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

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

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

To conclude, ANSIBLEMETRICS can already compute several code metrics that can be generalized to other languages. Although, at the current stage, it does not provide any further analysis.

A.5. SUMMARY

This chapter described ANSIBLEMETRICS, a tool we created to measure the software quality of Ansible blueprints. ANSIBLEMETRICS aims to step forward to support DevOps engineers when developing and maintaining infrastructure code.

It illustrated its architecture and usage examples through a command-line and graphical user interface. Although a GUI is available only as a VS Code extension, it can be easily integrated into other environments. More specifically, a version is planned for the Eclipse Che.[12] Indeed, starting with Eclipse Che 7.15, VS Code extensions can be effortlessly installed to extend the functionality of a Che workspace.[13]

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

We are confident that the proposed tool will help develop measurement models for the quality of Ansible blueprints and IaC. We also deem the tool appropriate and usable in practice, as validated by the industrial partners of the project RADON.

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

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

B

RepoMiner: a Modular Python Framework to Mine Software Repositories for Defect Prediction

Mining software repositories (MSR) is common for researchers to collect and empirically investigate the rich data available in software repositories to uncover interesting and actionable insights about software systems and projects.[1] These insights are particularly useful in software engineering research as they identify best practices to improve software quality, resource allocation, bug prediction, and more (Chaturvedi et al. 2013).

Among these, predicting classes prone to defects using prediction models based on machine learning is a mature research area that strongly relies on mining techniques and tools. Building defect prediction models typically consist of mining software archives such as version control and issue tracking systems to generate instances from software components, such as classes and methods; instances are characterized using metrics. For example, process metrics measure aspects of the development process, such as the number of files committed together, or product metrics measure the structural characteristics of source files, such as the number of code lines. Afterward, software components are labeled as "failure-prone" or "neutral" and used as ground truth by machine learning classifiers to learn which metrics discriminate and predict defects in a specific component. Unfortunately, labeling is challenging and time-consuming.

To tackle this problem, we implemented RepoMiner, a modular framework that combines failure data acquisition and metrics extraction to help researchers mine software repositories and create datasets for defect prediction. RepoMiner does not reinvent the wheel by implementing yet another mining framework. Instead, it builds

[1] http://www.msrconf.org/

on top of the well-known Python framework *PyDriller* (Spadini et al. 2018) to analyze a repository's history and provides APIs to

- Identify commits that take action to remove a defect, i.e., defect-fixing commits.
- Identify failure-prone software components across the project's history and label them failure-prone or neutral.
- Extract process and language-specific source code metrics from those components and generate a dataset of observations to train defect prediction models.

Researchers can manipulate the extracted data programmatically and extend the framework to support new languages. The tool is open-source and available on GitHub[2] and PyPI[3].

This chapter presents the labeling and metrics extraction implementation of REPOMINER and gives examples of its application. The tool is part of the framework presented in Chapter 4 and was instantiated in our previous work on Infrastructure-as-Code (IaC) defect prediction described in Chapter 5 to collect failure-prone data that could be used to train defect prediction models for Ansible.

The remainder of this chapter is structured as follows. Appendix B.1 analyzes tools similar to REPOMINER and highlights the main differences. Appendix B.2 describes REPOMINER's architecture. Appendix B.3 illustrates its usage in Infrastructure-as-Code (IaC) defect prediction and shows possible extensions to new languages. Appendix B.4 outlooks current limitations and possible extensions.

B.1. RELATION WITH SIMILAR TOOLS

Many researchers use already developed tools for data extraction from software engineering repositories, pattern finding, learning, and prediction. Nevertheless, they often implement custom scripts for their mining tasks instead of using the available tools. For example, (Chaturvedi et al. 2013) surveyed MSR-related papers to study the usage and purpose of different tools. Among them, we found no tools supporting automatic labeling of failure-prone software components and metrics extraction from those components for defect prediction. Kuhn 2009 presents a lexical approach to retrieve labels from source code automatically. However, it is only a prototype implementation that obtains labels describing components to compare rather than identify those components deemed failure-prone. The most similar to REPOMINER is ALITHEIA CORE, an extensible platform for performing large-scale software engineering and repository mining studies (Gousios et al. 2009). Its capability to be extended by plug-ins that calculate process and product metrics is particularly interesting. Nevertheless, it does not enable mining failure-prone components and is archived.

B.2. CHARACTERISTICS AND OUTLINE

REPOMINER consists of two modules: *mining* and *metrics*; each module implements a *base* component that provides language-agnostic functionalities for that module. Base

[2]`https://github.com/radon-h2020/radon-repository-miner`
[3]`https://pypi.org/project/repository-miner//`

Category	Description
Conditional	Defects due to erroneous logic or conditional values used to execute one or multiple brancehs of an IaC script.
Configuration Data	Defects due to erroneous configuration data that reside in IaC scripts.
Dependency	Defects that occur when executing an IaC script depends upon an artifact that is either missing or incorrectly specified.
Documentation	Defects that occur when incorrect information about IaC scripts are specified in source code comments, in maintenance notes, and in documentation files such as README files.
Idempotency	Defects that violate the idempotency property for IaC scripts. Idempotency ensures that even after multiple execution the provisioned systems's environment is exactly the same as it was after the first execution of the relevant IaC scripts.
Security	Defects that violate confidentiality, integrity or availability for the provisioned system.
Service	Defects related to improper provisioning and inadequate availability of computing services, such as load balancing services and monitoring services.
Syntax	Defects related to syntax in IaC scripts.

Table B.1: The defect taxonomy for IaC defined in Rahman, Farhana, Parnin, et al. 2020.

components implement the classes *mining.BaseMiner* and *mining.FixingCommitClassifier* for repository mining and defect-fixing commits categorization; they extend *PyDriller* (Spadini et al. 2018) to analyze the project history and identify *defect-fixing* and *bug-introducing* commits and *failure-prone* files needed for the analysis. In addition, the class *metrics.BaseMetricsExtractor* extracts metrics from the mined scripts; it uses *PyDriller* to extract process metrics and ad-hoc external libraries to extract product metrics from the mined files.

The base components can then be extended to support a specific language. The following sections describe these components in detail.

B.2.1. BASEMINER

BaseMiner is an abstract class that must be extended to mine repositories for a specific language. Mandatory inputs are the *URL* to a remote git repository (e.g., a GitHub or GitLab repository) and the *path to clone* the repository on the disk. Optional input is the *branch* to analyze, e.g., *main* or *master*.

Then, the mining consists of the three methods below, to be executed sequentially:

1. **get_fixing_commits()** – Identifies *defect-fixing* commits and related defect(s) using the rules defined in Rahman, Farhana, Parnin, et al. 2020. Defects – described in Table B.1 – relate to "conditionals", "configuration data", "dependencies", "documentation", "idempotency", "security", "service", and "syntax" issues.

2. **get_fixed_files()** – Identifies files modified in the defect-fixing commits, called *fixed files*, and their *bug-introducing* commit using the *SZZ* algorithm[4] (Kim et al. 2006). A fixed file consists of a *filepath* and the hashes of its *defect-introducing commit* and *defect-fixing commit*.

3. **label()** – Tags as *failure-prone* all the snapshots of a fixed file between its bug-introducing commit (inclusive) and its defect-fixing commit. A tagged snapshot,

[4]As implemented in *PyDriller* at release ≥ 2.0.

Listing 15 Usage template of `BaseMiner`.

```
from repominer.mining.base import BaseMiner

miner = BaseMiner(url_to_repo='github.com/radon-h2020/radon-defuse', clone_repo_to='/tmp')

miner.get_fixing_commits()
miner.get_fixed_files()
failure_prone_files = miner.label()
```

i.e., a failure-prone file, consists of a *filepath*, a *commit hash*, and the hash of its *defect-fixing commit*.

BaseMiner has an additional method called **ignore_file()** that the sub-classes *must* implement to filter commits and files for a given language. That method should return True when a file has to be ignored, based on its path, extension, or content. For example, the method will ignore all non *.py* files for Python-based repositories. Listing 15 shows a usage template of *BaseMiner*.

B.2.2. FixingCommitClassifier

FixingCommitClassifier is an abstract class that categorizes fixing commits based on the defects categories mentioned in Appendix B.2.1. It provides base implementations for each category, although some have to be overridden by sub-classes due to the language. For example, the methods **fixes_configuration_data()** and **fixes_idempotency()** relate to defect categories specific to configuration management and infrastructure provisioning and do not apply to application code languages. By contrast, the method **fixes_dependency()** is common in many languages, although its implementation may change depending on the syntax. Therefore, concrete *ExtendedFixingCommitClassifier* classes should override some methods according to the language.

B.2.3. BaseMetricsExtractor

BaseMetricsExtractor is an abstract class that extracts process and product metrics from the collected files and creates a dataset of *failure-prone* and *neutral* observations ready for defect prediction. Mandatory input is the *path* to the local or remote git repository. Optional inputs are

- the *path to clone* the repository on the disk, if *path* is a URL;
- *when* to extract metrics: at each release or each commit.

The entry point is the method **extract()**, which requires a list of failure-prone files and the type of metrics to extract, i.e., *process*, *product* or *delta*, or group thereof. Then, the methods **get_process_metrics()** and **get_product_metrics()** are used to extract process and product metrics, respectively. It is worth mentioning that because product metrics are language-dependant, the classes that extract source code metrics for a specific language must override the method **get_process_metrics()**.

Finally, similarly to *BaseMiner*, *ExtendedMetricsExtractor* must implement the method **ignore_file()** to filter only the files written in the language relevant targeted by the class.

Listing 16 shows a usage template of *BaseMiner*.

Listing 16 Usage template of `BaseMetrics`.

```
from repominer.metrics.base import BaseMetrics

bm = BaseMetrics(url_to_repo='github.com/radon-h2020/radon-defuse', clone_repo_to='/tmp',
↪  at='release')

labeled_files = [...] # labeled files from BaseMiner.label()
bm.extract(labeled_files, process=True, product=True, delta=True)
print(bm.dataset())
```

Method	Min	Mean	Std	Median	Max
AnsibleMiner.get_fixing_commits()	10	110	170	50	1170
AnsibleMiner.get_fixed_files()	0	30	40	10	300
AnsibleMiner.label()	0	40	90	10	740
AnsibleMetricsExtractor.extract()	20	450	1,400	170	13,540
Total seconds	30	620	1,615	250	15,030

Table B.2: Statistics on execution time in seconds, rounded to the nearest ten.

B.3. VALIDATION

In the study presented in Chapter 5, we ran REPOMINER on 104 Ansible projects to collect failure-prone data that could be used to train defect prediction models for Ansible. There, REPOMINER identified almost five thousand fixing commits (a median of 26 per project) and approximately 14,000 defective observations (a median of 530 per project and a mean of three observations per release).

Table B.2 reports the execution time of REPOMINER on the 104 projects. Performance varied from a median of 50 seconds for identifying fixing commits to ten seconds for labeling failure-prone files and three minutes for extracting process and product metrics. In some cases, *extract_metrics()* ran up to three hours before completing. For example, for the repository *redhat-openstack/infrared*, where REPOMINER collected a dataset of approximately 76,000 observations over 6,000 commits across 145 releases. There, product metrics had to be extracted for every observation, and process metrics had to be extracted for every release.

We measured the soundness of REPOMINER in identifying *defect-fixing* commits of Ansible files. We uniformly selected and manually assessed a statistically relevant subsample of 357 fixing commits and obtained a precision of 74%. Next, we applied the same procedure to evaluate its precision in identifying *bug-introducing* commits. We validated 354 bug-introducing commits and obtained a precision of 84%. More details about the validation procedure can be found in Chapter 5.

B.4. LIMITATIONS AND EXTENSIONS

REPOMINER identifies fixing commits based on the defect taxonomy proposed by Rahman, Farhana, Parnin, et al. 2020, which focuses on Infrastructure-as-Code (IaC). However, other approaches could be implemented, combined, and compared. For example, Babii et al. 2021 proposed BOHR, a repository where researchers can contribute heuristics to label Software Engineering data, such as bug-fixing commits.[5] BOHR uses weak supervision techniques to combine these heuristics and train classifiers operating on Software Engineering data. Ideally, many similar heuristics might be suitable and easily integrated into REPOMINER because both tools are written in Python.

Furthermore, although REPOMINER was initially developed to support Ansible and Tosca, it was designed to be easily extendable to additional IaC languages such as Chef and Puppet and application code languages such as Python and Java.

B.5. SUMMARY

This chapter presented REPOMINER, a novel tool to collect *defect-fixing* commits, *failure-prone* files, and their process and source code metrics. It aims to make a step forward to support research when mining software repositories to ease the creation of datasets for defect prediction studies. REPOMINER is designed to work on *git* repositories and potentially could be used for any infrastructure and application code language.

It illustrated the tool architecture and provided examples of its programming interface. Nevertheless, further interaction modalities with the tool, such as a web application, are already included in DEFUSE (Chapter 7). In addition, we received encouraging feedback from the industrial partners within the project RADON. Therefore, we hope and plan to enlarge the user community around REPOMINER through their collaboration and dissemination. We are confident that the proposed tool will help researchers and practitioners collect meaningful data to feed defect prediction models either in the context of infrastructure or application code.

[5]https://github.com/giganticode/bohr

Bibliography

Aniche, Bavota, Treude, Gerosa, and van Deursen (2018). “Code Smells for Model-View-Controller Architectures”. In: *Empirical Software Engineering*.

Arisholm, Briand, and Johannessen (2010). “A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models”. In: *Journal of Systems and Software*.

Azeem, Palomba, Shi, and Wang (2019). “Machine Learning Techniques for Code Smell Detection: A Systematic Literature Review and Meta-analysis”. In: *Information and Software Technology*.

Babii, Prenner, Stricker, Karmakar, Janes, and Robbes (2021). “Mining Software Repositories with a Collaborative Heuristic Repository”. In: *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results*.

Baeza-Yates and Ribeiro-Neto (1999). *Modern Information Retrieval*. ACM press New York.

Baresi, Quattrocchi, Tamburri, and Van Den HeuvelJan (2020). “Automated Quality Assessment of Incident Tickets for Smart Service Continuity”. In: *Service-Oriented Computing*. Springer International Publishing.

Basiri, Ali et al. (2016). “Chaos Engineering.” In: *IEEE Software* 33.3, pp. 35–41. URL: http://dblp.uni-trier.de/db/journals/software/software33.html#BasiriBRHKRR16.

Batista, Prati, and Monard (2004). “A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data”. In: *SIGKDD Explor. Newsl.*

Beck (2003). *Test-driven Development: by Example*. Addison-Wesley Professional.

Bent, van der, Hage, Visser, and Gousios (2018). “How Good is Your Puppet? An Empirically Defined and Validated Quality Model for Puppet”. In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*.

Binz, Breitenbücher, Kopp, and Leymann (2014). “TOSCA: Portable Automated Deployment and Management of Cloud Applications”. In: *Advanced Web Services*. Springer.

Bowes, Hall, and Petrić (2018). “Software Defect Prediction: Do Different Classifiers Find the Same Defects?” In: *Software Quality Journal*.

Boyd, Eng, and Page (2013). “Area under the Precision-Recall Curve: Point Estimates and Confidence Intervals”. In: *Machine Learning and Knowledge Discovery in Databases*.

BroadGroup (2019). *Salesforce customers still experiencing issues caused by 15-hour outage*. URL: https://data-economy.com/salesforce-customers-still-experiencing-issues-caused-by-15-hour-outage/.

Caballer, Miguel, Sahdev Zala, Alvaro Lopez Garcia, German Molto, Pablo Orviz Fernandez, and Mathieu Velten (2018). “Orchestrating Complex Application Architectures in Heterogeneous Clouds”. In: *Journal of Grid Computing* 16.1, pp. 3–18.

Catolino, Palomba, Fontana, De Lucia, Zaidman, and Ferrucci (2020). "Improving Change Prediction Models with Code Smell-related Information". In: *Empirical Software Engineering.*

Chambers (2018). *Graphical Methods for Data Analysis.* CRC Press.

Chaturvedi, Sing, and Singh (2013). "Tools in Mining Software Repositories". In: *International Conference on Computational Science and Its Applications.* IEEE.

Chidamber and Kemerer (1994). "A Metrics Suite for Object Oriented Design". In: *Transactions on Software Engineering.*

Cliff (1993). "Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions". In: *Psychological Bulletin.*

Cochran (1977). *Sampling Techniques.* John Wiley & Sons.

Cohen (1960). "A Coefficient of Agreement for Nominal Scales". In: *Educational and Psychological Measurement.*

— (1988). "The Effect Size Index: d". In: *Statistical power analysis for the behavioral sciences.*

ComputerWeekly.com (2016). *HSBC suffers major online banking failure.* URL: https://www.computerweekly.com/news/4500269864/HSBC-suffers-major-online-banking-failure.

— (2019). *TSB blames IT disaster for huge losses.* URL: https://www.computerweekly.com/news/252456881/TSB-blames-IT-disaster-on-huge-losses.

Conover (1998). *Practical Nonparametric Statistics.* John Wiley & Sons.

D'Ambros, Lanza, and Robbes (2012a). "Evaluating Defect Prediction Approaches: a Benchmark and an Extensive Comparison". In: *Empirical Software Engineering.*

— (2012b). "Evaluating Defect Prediction Approaches: a Benchmark and an Extensive Comparison". In: *Empirical Software Engineering.*

— (2012c). "Evaluating Defect Prediction Approaches: a Benchmark and an Extensive Comparison". In: *Empirical Software Engineering.*

Dalla Palma, Di Nucci, Palomba, and Tamburri (2020). "Toward a Catalog of Software Quality Metrics for Infrastructure Code". In: *Journal of Systems and Software.*

Dalla Palma, Di Nucci, and Tamburri (2020). "AnsibleMetrics: a Python Library for Measuring Infrastructure-as-Code Blueprints in Ansible". In: *SoftwareX.*

Dalla Palma, Stefano, Gemma Catolino, Dario Di Nucci, Damian Andrew Tamburri, and Willem-Jan van den Heuvel (2023). "Go Serverless With RADON! A Practical DevOps Experience Report". In: *IEEE Software* 40.2, pp. 80–89. DOI: 10.1109/MS.2022.3170153.

DataCenterKnowledge (2017). *AWS Outage that Broke the Internet Caused by Mistyped Command.* URL: https://www.datacenterknowledge.com/archives/2017/03/02/aws-outage-that-broke-the-internet-caused-by-mistyped-command.

Di Nucci, Palomba, De Rosa, Bavota, Oliveto, and De Lucia (2018). "A Developer Centered Bug Prediction Model". In: *Transactions on Software Engineering.*

Di Nucci, Palomba, Tamburri, Serebrenik, and De Lucia (2018). "Detecting Code Smells Using Machine Learning Techniques: Are We There Yet?" In: *Proceedings of the International Conference on Software Analysis, Evolution and Eeengineering.*

Falessi, Huang, Narayana, Thai, and Turhan (2020). “On the Need of Preserving Order of Data When Validating Within-project Defect Classifiers”. In: *Empirical Software Engineering*.

Fenton and Bieman (2014). *Software Metrics: a Rigorous and Practical Approach*. CRC press.

Filzmoser, Reimann, and Garrett (2004). *A Multivariate Outlier Detection Method*.

Fontana, Mantyla, Zanoni, and Marino (2016). “Comparing and Experimenting Machine Learning Techniques for Code Smell Detection”. In: *Empirical Software Engineering*.

Fowler (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

Fu and Menzies (2017). “Revisiting Unsupervised Learning for Defect Prediction”. In: *Proceedings of the Joint Meeting on Foundations of Software Engineering*.

Fu, Menzies, and Shen (2016). “Tuning for Software Analytics: Is it Really Necessary?” In: *Information and Software Technology*.

GeekWire (2015). *Starbucks lost millions in sales because of a ‘system refresh’ computer problem*. URL: https://www.geekwire.com/2015/starbucks-lost-millions-in-sales-because-of-a-system-refresh-computer-problem/.

Goadrich, Oliphant, and Shavlik (2006). “Gleaner: Creating Ensembles of First-order Clauses to Improve Recall-Precision Curves”. In: *Machine Learning*.

Gousios and Spinellis (2009). “A Platform for Software Engineering Research”. In: *Proceedings of the International Working Conference on Mining Software Repositories*.

Guerriero, Garriga, Tamburri, and Palomba (2019). “Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry”. In: *Proceedings of the International Conference on Software Maintenance and Evolution*.

Guggulothu and Moiz (2019). “Detection of Shotgun Surgery and Message Chain Code Smells Using Machine Learning Techniques”. In: *International Journal of Rough Sets and Data Analysis*.

Hall, Beecham, Bowes, Gray, and Counsell (2012). “A Systematic Literature Review on Fault Prediction Performance in Software Engineering”. In: *Transactions on Software Engineering*.

Hosseini, Turhan, and Gunarathna (2019). “A Systematic Literature Review and Meta-Analysis on Cross Project Defect Prediction”. In: *Transactions on Software Engineering*.

IEEE (2010). “IEEE Standard Classification for Software Anomalies - Redline”. In: *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993) - Redline*.

Jarschel (2013). *Network Function Virtualization: Towards the Commoditization of Middle Boxes*.

Jiang and Adams (2015). “Co-evolution of Infrastructure and Source Code - An Empirical Study”. In: *Working Conference on Mining Software Repositories*.

Kalliamvakou, Gousios, Blincoe, Singer, German, and Damian (2014). “The Promises and Perils of Mining GitHub”. In: *Proceedings of the Working Conference on Mining Software Repositories*.

Kampenes, Dybå, Hannay, and Sjøberg (2007). “A Systematic Review of Effect Size in Software Engineering Experiments”. In: *Information and Software Technology*.

Keating (2015). *Mastering Ansible*. Packt Publishing Ltd.

Khomh, Di Penta, Guéhéneuc, and Antoniol (2012). “An Exploratory Study of the Impact of Antipatterns on Class- and Fault-proneness”. In: *Empirical Software Engineering*.

Kim, Zimmermann, Pan, and Jr. Whitehead (2006). “Automatic Identification of Bug-Introducing Changes”. In: *Proceedings of the International Conference on Automated Software Engineering*.

Kotrlik, Higgins, and Bartlett (2001). “Organizational Research: Determining Appropriate Sample Size in Survey Research”. In: *Information Technology, Learning, and Performance Journal*.

Kuhn (2009). “Automatic Labeling of Software Components and Their Evolution Using Log-likelihood Ratio of Word Frequencies in Source Code”. In: *IEEE International Working Conference on Mining Software Repositories*.

Lemaître, Nogueira, and Aridas (2017). “Imbalanced-learn: a Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning”. In: *Journal of Machine Learning Research*.

Lewis and Hepburn (2010). “Open Card Sorting and Factor Analysis: a Usability Case Study”. In: *The Electronic Library*.

Li and Shatnawi (2007). “An Empirical Study of the Bad Smells and Class Error Probability in the Post-release Object-oriented System Evolution”. In: *Journal of Systems and Software*.

Li, Shepperd, and Guo (2020). “A Systematic Review of Unsupervised Learning Techniques for Software Defect Prediction”. In: *Information and Software Technology*.

Lipton, Palma, Rutkowski, and Tamburri (2018). “TOSCA Solves Big Problems in the Cloud and Beyond!” In: *IEEE Cloud Computing*.

Mansfield and Helms (1982). “Detecting Multicollinearity”. In: *The American Statistician*.

Marinescu (2004). “Detection Strategies: Metrics-based Rules for Detecting Design Flaws”. In: *Proceedings of the International Conference on Software Maintenance*.

— (2005). “Measurement and Quality in Object-oriented Design”. In: *Proceedings of the International Conference on Software Maintenance*.

Matthews (1975). “Comparison of the Predicted and Observed Secondary Structure of T4 Phage Lysozyme”. In: *Biochimica et Biophysica Acta (BBA)-Protein Structure*.

McLachlan (1999). “Mahalanobis Distance”. In: *Resonance*.

Morris (2016). *Infrastructure as Code*. O’Reilly Media, Inc.

Moser, Pedrycz, and Succi (2008). “A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction”. In: *Proceedings of the International Conference on Software Engineering*.

Munaiah, Kroh, Cabrey, and Nagappan (2017). “Curating GitHub for Engineered Software Projects”. In: *Empirical Software Engineering*.

Münch and Schmid (2013). *Perspectives on the Future of Software Engineering*. Springer.

Nam and Kim (2015). “Clami: Defect Prediction on Unlabeled Datasets”. In: *Proceedings of the International Conference on Automated Software Engineering*.

Neter, Kutner, Nachtsheim, and Wasserman (1996). *Applied Linear Statistical Models*. Irwin Chicago.

Neto, Da Costa, and Kulesza (2018). “The Impact of Refactoring Changes on the SZZ Algorithm: an Empirical Study”. In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*.

Nuñez-Varela, Perez-Gonzalez, Martinez-Perez, and Soubervielle-Montalvo (2017). "Source Code Metrics: A Systematic Mapping Study". In: *Journal of Systems and Software*.

Olbrich, Cruzes, and Sjøberg (2010). "Are all Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of Three Open Source Systems". In: *Proceedings of the International Conference on Software Maintenance*.

PaloaltoNetworks Research (2022). *Infrastructure as Code: Putting the Sec in DevOps*. https://www.paloaltonetworks.com/resources/research/cloud-threat-report-spring-2020. Accessed June 2022.

Palomba, Bavota, Di Penta, Fasano, Oliveto, and De Lucia (2018). "On the Diffuseness and the Impact on Maintainability of Code Smells: a Large Scale Empirical Investigation". In: *Empirical Software Engineering*.

Palomba, Bavota, Di Penta, Oliveto, Poshyvanyk, and De Lucia (2014). "Mining Version Histories for Detecting Code Smells". In: *Transactions on Software Engineering*.

Palomba, Panichella, De Lucia, Oliveto, and Zaidman (2016). "A Textual-based Technique for Smell Detection". In: *Proceedings of the International Conference on Program Comprehension*.

Paramshetti and Phalke (2014). "Survey on Software Defect Prediction Using Machine Learning Techniques". In: *International Journal of Science and Research*.

Pedregosa et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research*.

Rahman and Devanbu (2013). "How, and Why, Process Metrics are Better". In: *Proceedings of the International Conference on Software Engineering*.

Rahman, Farhana, Parnin, and Williams (2020). "Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts". In: *Proceedings of the International Conference on Software Engineering*.

Rahman, Farhana, and Williams (2020). "The 'as code' Activities: Development Antipatterns for Infrastructure as Code". In: *Empirical Software Engineering*.

Rahman, Mahdavi-Hezaveh, and Williams (2019). "A Systematic Mapping Study of Infrastructure as Code Research". In: *Information and Software Technology*.

Rahman, Parnin, and Williams (2019). "The Seven Sins: Security Smells in Infrastructure as Code Scripts". In: *International Conference on Software Engineering*.

Rahman, Partho, Morrison, and Williams (2018). "What Questions Do Programmers Ask about Configuration as Code?" In: *Proceedings of the International Workshop on Rapid Continuous Software Engineering*.

Rahman, Posnett, Herraiz, and Devanbu (2013). "Sample Size vs. Bias in Defect Prediction". In: *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*.

Rahman and Williams (2018). "Characterizing Defective Configuration Scripts Used for Continuous Deployment". In: *Proceedings of the International Conference on Software Testing, Verification and Validation*.

— (2019). "Source Code Properties of Defective Infrastructure as Code Scripts". In: *Information and Software Technology*.

Rousseeuw (1987). "Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis". In: *Journal of Computational and Applied Mathematics*.

Sadiku, Musa, and Momoh (2014). "Cloud Computing: Opportunities and Challenges". In: *IEEE Potentials*.

Schwarz, Steffens, and Lichter (2018). “Code Smells in Infrastructure as Code”. In: *Proceedings of the International Conference on the Quality of Information and Communications Technology.*

Settles (2009). “Active Learning Literature Survey”. In.

Sharma, Fragkoulis, and Spinellis (2016). “Does Your Configuration Code Smell?” In: *Proceedings of the International Conference on Mining Software Repositories.* Association for Computing Machinery.

Sharma and Spinellis (2018). “A Survey on Software Smells”. In: *Journal of Systems and Software.*

Sobrinho, De Lucia, and Maia (2018). “A Systematic Literature Review on Bad Smells — 5 W’s: Which, When, What, Who, Where”. In: *Transactions on Software Engineering.*

Soldani, Barani, Tafazolli, Manzalini, and Chih-Lin (2015). “Software Defined 5G Networks for Anything as a Service”. In: *IEEE Communications Magazine.*

Spadini, Aniche, and Bacchelli (2018). “PyDriller: Python Framework for Mining Software Repositories”. In: *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).*

Tantithamthavorn, McIntosh, Hassan, and Matsumoto (2016). “Automated Parameter Optimization of Classification Techniques for Defect Prediction Models”. In: *Proceedings of the International Conference on Software Engineering.*

TheGuardian (2018). *O2 poised to receive millions from Ericsson over software failure.* URL: https://www.theguardian.com/business/2018/dec/09/o2-poised-to-receive-millions-from-ericsson-over-software-failure.

— (2019). *Passenger anger as tens of thousands hit by BA systems failure.* URL: https://www.theguardian.com/business/2019/aug/07/british-airways-it-glitch-causes-disruption-for-passengers-delays.

Tufano et al. (2017). “When and Why your Code Starts to Smell Bad (and Whether the Smells Go Away)”. In: *Transactions on Software Engineering.*

Weisstein (2004). “Bonferroni Correction”. In: *https://mathworld. wolfram. com/.*

Wilcoxon (1992). “Individual Comparisons by Ranking Methods”. In: *Breakthroughs in Statistics.*

Xu et al. (2021). “A Comprehensive Comparative Study of Clustering-based Unsupervised Defect Prediction Models”. In: *Journal of Systems and Software.*

Zhang (2009). “An Investigation of the Relationships Between Lines of Code and Defects”. In: *International Conference on Software Maintenance.*

Zhang, Zheng, Zou, and Hassan (2016). “Cross-project Defect Prediction Using a Connectivity-based Unsupervised Classifier”. In: *Proceedings of the International Conference on Software Engineering.*

Zhiqiang, Xiao-Yuan, and Xiaoke (2018). “Progress on Approaches to Software Defect Prediction”. In: *IET Software.*

GLOSSARY

Blob blueprint A too large and complex topology that should be modularized further. 10, 63

blueprint A description of a software-defined infrastructure, which can be used as a template for code generators for its automated provisioning, deployment, and run-time management. It specifies infrastructure's components – i.e., hardware, software, and networks –, their configuration, and their relationships and capabilities, along with policies for provisioning, deployment, and run-time management. 2, 26, 68, 94

code smell Lousy coding practice affecting software maintainability. 4, 20, 63

defect Imperfection or deficiency causing the code not to meet its requirements or specifications. 4, 17, 20, 84, 112

defect-fixing commit Commit that takes an action related to a defect. 17, 38, 86, 113

defect-introducing commit Commit that contributed to introduce a defect. 17, 39, 113

failure-prone script Script that presents defects and needs to be either repaired or replaced, as opposite of a *neutral* script. 17, 35, 38, 45, 83, 93

Infrastructure-as-Code Machine-readable, configuration code to automatically provision infrastructures. 5, 13, 23, 35, 83, 91, 101, 112, 115

module In Ansible, a module is a discrete unit of code that can be used from the command line or in a playbook task to describe or change the system state. 14, 24

node In TOSCA, it represents a service component in the topology. 15, 65

play What a playbook is composed of. Each play executes part of the overall goal of the playbook, running one or more tasks. 14, 24, 102

playbook Offers a repeatable, re-usable, simple configuration management and multi-machine deployment system, one that is well suited to deploying complex applications. 14, 24, 92

process metric A metric related to the development process, for example the number of developers who changed a file, the total number of added and removed lines, or the number of files committed together. 6, 18, 33, 35, 38, 45, 93, 111

product metric A metrics related to the structural properties of the source code, for example the number of lines of code and methods. 6, 18, 33, 38, 45, 93, 111

provisioning The process of setting up IT infrastructure to allow it to provide new services to its users. 1

relationship In TOSCA, it represent the relationship that connects and structures nodes into the topology. 16, 65

role A role let you automatically load related vars, files, tasks, handlers, and other Ansible artifacts based on a known file structure. After you group your content in a role, you can easily reuse it and share it with other users. 28

task Call to an Ansible module. 14, 24, 92, 102

technical debt The implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer. 6

template In TOSCA, it forms the cloud service's topology using a TOSCA type. It defines how the respective type is instantiated for use in the application. A template is conceptually comparable to a concrete class. 16, 65

type In TOSCA, it defines a reusable entity that defines the semantics of the node or relationship (e.g., properties, attributes, requirements, capabilities, interfaces). A type is conceptually comparable to an abstract class. 16, 65