ABSTRACT

RAHMAN, AKOND ASHFAQUE UR. Anti-patterns in Infrastructure as Code. (Under the direction of Laurie Williams).

In continuous deployment, infrastructure as code (IaC) scripts are used by practitioners to create and manage an automated deployment pipeline that enables information technology (IT) organizations to release their software changes rapidly at scale. Low quality IaC scripts can have serious consequences, potentially leading to wide-spread system outages and service discrepancies. By systematically identifying which characteristics are correlated with low quality IaC scripts, we can identify anti-patterns i.e. recurring practices with negative consequences, which may help practitioners to take informed actions in creating and maintaining defect-free IaC scripts. *The goal of this thesis is to help practitioners in increasing quality of IaC scripts by identifying development and security anti-patterns in the development of infrastructure as code scripts.* Using open source repositories, we conduct five research studies and identify (i) defect categories in IaC scripts; (ii) operations that characterize defective IaC scripts; (iii) code properties that correlate with defective IaC scripts; (iv) development anti-patterns for IaC scripts; and (v) security anti-patterns in IaC scripts that are indicative of security weaknesses.

From our conducted empirical studies, we observe defect category distribution of IaC scripts are different to that of general purpose programming languages. We observe three operations that characterize defective IaC scripts, namely file system operations, infrastructure provisioning, and user account management. We identify 10 source code properties that correlate with defective scripts, of which size and hard-coded strings as configuration values show the strongest correlation with defective scripts. We identify 13 development anti-patterns that are development activities, which correlate with defective scripts. We also identify 9 security anti-patterns i.e. coding patterns that are indicative of security weaknesses. We hope outcomes of the thesis i.e. findings, tools, and datasets will facilitate further research in the area of IaC.

Anti-patterns in Infrastructure as Code

by
Akond Ashfaque Ur Rahman

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2019

APPROVED BY:

DEDICATION

To my mother, Dr. Parveen Akhter, the greatest mentor of my life, whose teachings and endless sacrifices inspired me to all my success.

ACKNOWLEDGEMENTS

First of all I thank the Almighty. Next, I thank my parents especially my mother Dr. Parveen Akhter for her endless sacrifices. I thank my brothers Partho and Babu, and my wife Effat for their support and inspiration. I want to thank my adviser, Laurie Williams, one of the nicest human beings I have met in my life. Special thanks to my dissertation committee members: Chris Parnin, Tim Menzies, and Jonathan Stallings. In particular, my research and technical skills expanded significantly after I worked with Tim and Chris on multiple research projects. I thank my colleagues from the RealSearch, Alt-Code, RAISE, WSPR, and DLF research group members at North Carolina State University from whom I learned a lot. I want to thank my Bangladeshi friends from Bangladesh University of Engineering and Technology, University of Connecticut, and North Carolina State University for their support. Special thanks to Tauhidur Rahman of University of Alabama Huntsville, who guided me throughout my PhD career as an unofficial mentor and a brother. I also thank my colleagues at ABB Corporate Research, RedHat, IBM, and IBM Research who helped me learn about industry perspectives of software engineering research. During my PhD tenure I authored/co-authored 23 publications, with 23 co-authors. I thank all my co-authors whose feedback made my research better. The National Security Agency (NSA)'s Science of Security Lablet at North Carolina State University funded my PhD research.

Parts of this dissertation were published as a technical report and as papers at the Journal of Information and Software Technology (IST) 2018 [Rah18a] (Chapter 2); International Conference on Software Testing, Verification, and Validation (ICST) 2018 [RW18] (Chapter 5); IST 2019 [RW19] (Chapter 6); International Conference on Software Testing (ICSE) 2019 [Rah19b] (Chapter 8). Chapter 7 is currently under review at the Empirical Software Engineering journal. Part of Chapter 8 is currently under review at the IEEE Transactions on Software Engineering journal. I would like to thank all the anonymous reviewers for their valuable feedback on my papers, which made the dissertation better. At last but not the least, I express my deepest gratitude to the ACM SIGSOFT Community for honoring me with the ACM SIGSOFT Doctoral Dissertation Award at ICSE 2018 and the ACM SIGSOFT Distinguished Paper Award at ICSE 2019.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1 INTRODUCTION

Continuous deployment is the process of rapidly deploying software or services automatically to end-users [Rah15]. The practice of infrastructure as code (IaC) scripts is essential to implement an automated deployment pipeline, which facilitates continuous deployment [HF10]. Information technology (IT) organizations, such as Netflix [1], Ambit Energy [2], and Wikimedia Commons [3], use IaC scripts to automatically manage their software dependencies, and construct automated deployment pipelines [HF10] [Par17]. Commercial IaC tools, such as Ansible [4], Chef![5], and Puppet [6], provide multiple utilities to construct automated deployment pipelines. Use of IaC scripts has helped IT organizations to increase their deployment frequency. For example, Ambit Energy, uses IaC scripts to increase their deployment frequency by a factor of 1,200 [7].

IaC is the practice of specifying computing and software deployment infrastructure using code. Practitioners also use these scripts to deploy software services, such as database, data analytics and monitoring, on cloud instances, and to control which groups of users will get access to those services [Lab17] [MJ11]. Use of these scripts facilitate rapid deployment, as repetitive, manual system administration work is replaced by automation. The prefix 'as code' in IaC corresponds

[1]https://www.netflix.com/
[2]https://www.ambitenergy.com/
[3]https://commons.wikimedia.org/wiki/Main_Page
[4]https://www.ansible.com/
[5]https://www.chef.io/
[6]https://puppet.com/
[7]https://puppet.com/resources/case-study/ambit-energy

to the practice of treating IaC scripts as software source code [HF10] [JA15]. Practitioners apply the traditional software engineering practices, such as code review and version control, for IaC scripts [Par17][HF10]. Treating IaC scripts as software source code can provide multiple benefits for IT organizations, for example, achieving the capability to make frequent changes and improvements to the infrastructure at scale, and increased visibility within the organization as both development and operations teams can access these scripts [HF10] [Par17].

However, similar to software source code, IaC scripts can experience frequent churn, making these scripts susceptible to quality issues, such as defects [JA15] [Par17]. Figure 1.1 provides an example defect in an IaC script where a wrong package name is specified. Defects in IaC scripts can have serious consequences, as these scripts are associated in setting up and managing cloud-based infrastructure, and ensuring availability of software services.

Any defect in a script can propagate at scale, leading to service discrepancies. For example on January 2017, execution of a defective IaC script erased home directories of around 270 users in cloud instances maintained by Wikimedia [8]. The above-mentioned evidence demonstrated in real-world, and research studies motivate us to systematically study quality issues of IaC scripts in the form of anti-patterns. Anti-patterns in software engineering correspond to practices that may have negative consequences [Bro98a].

One strategy to mitigate defects is to identify source code properties and operations that relate with defective scripts. Practitioners can use the identified operations and properties for prioritizing inspection efforts in IaC scripts.

Another strategy is to identify anti-patterns applicable for IaC. Practitioners might be inadvertently implementing practices with negative consequences due to lack of knowledge, lack of experience, or applying a perceived good practice in the wrong context [Bro98a]. In prior work [Moh10] [Arn16], researchers have proposed and quantified a plethora of anti-patterns for software written in general purpose languages (GPLs) such as C, C++, and Java. For example, Brown proposed 14 implementation, 13 architecture, and 14 management anti-patterns for software source code. Fowler [Fow99] proposed a set of 22 code smells and structural smells in source code for object-oriented programming (OOP) languages. Moha et al. [Moh10] proposed and evaluated an anti-pattern detection tool called DETEX, and used DETEX to detect 15 latent code anti-patterns. In another work, Moha et al. [Moh06], proposed detection algorithms to detect design anti-patterns. However, these proposed anti-patterns may not be well-suited for IaC scripts, as IaC scripts use domain specific languages [Sha16a]. By identifying anti-patterns, we can provide actionable recommendations to practitioners and pinpoint characteristics that correlate with defects and violates security and privacy objectives. We take motivation from the above-mentioned literature and identify development and security anti-patterns.

In this thesis a development anti-pattern is a recurring development activity that relates with

[8]https://wikitech.wikimedia.org/wiki/Incident_documentation/20170118-Labs

```
{
    require => [                          Defect: wrong package
      Package['libmysql-java'],
      File['/var/lib/gerrit/review_site/lib'],
      Package['gerrit'],
    ],
    notify => Service['gerrit'],
}
```

Figure 1.1 Example of a defect included in the '825a073' commit, which is 460 lines long.

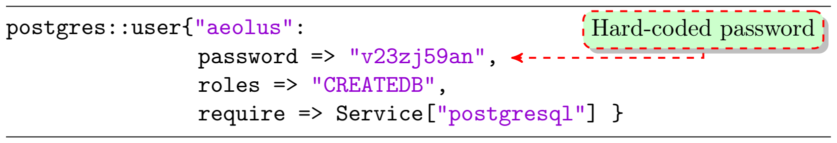
defective scripts, as determined by empirical analysis. We can identify development activities that relate with defective IaC scripts by mining open source software (OSS) repositories [AM16]. Let us consider Figure 1.1, which provides a code snippet from an actual 460-line long commit ('825a073') [9] that includes a defect. Reviewing large-sized commits, such as '825a073', is challenging [Mac18]. As a result, defects existent in large commits may not be detected during inspection. This anecdotal example suggests a relationship that may exist between the activity of submitting large commits and defects that appear in IaC scripts. By mining metrics from OSS repositories, we can identify recurring development activities, such as submitting large-sized commits, and investigate their relationship with defective IaC scripts. Our investigation can also reveal if not using 'as code' activities are related with defective IaC scripts.

Along with defects, IaC scripts can be susceptible to security anti-patterns. Security anti-patterns are recurring coding patterns that are indicative of security weakness. A security anti-pattern does not always lead to a security breach, but deserves attention and inspection. Let us consider Figure 1.2. In Figure 1.2, we present a Puppet code snippet extracted from the 'aeolus-configure' open source software (OSS) repository [10]. In this code snippet, we observe a hard-coded password using the 'password' attribute. A hard-coded string 'v23zj59an' is assigned as password for user 'aeolus'. Hard-coded passwords in software artifacts is considered as a software security weakness ('CWE-798: Use of Hard-coded Credentials') by Common Weakness Enumerator (CWE) [MIT18]. According to CWE [MIT18], "*If hard-coded passwords are used, it is almost certain that malicious users will gain access to the account in question*".

Existence and persistence of these security anti-patterns in IaC scripts leave the possibility of another programmer using these scripts with security anti-patterns, potentially propagating use of insecure coding practices. We hypothesize through systematic empirical analysis, we can identify security anti-patterns and the prevalence of the identified security anti-patterns.

The goal of this thesis is to help practitioners in increasing quality of IaC scripts by identifying development and security anti-patterns in the development of infrastructure as code scripts.

[9] https://github.com/Mirantis/puppet-manifests
[10] https://github.com/aeolusproject/aeolus-configure



```
postgres::user{"aeolus":
               password => "v23zj59an",
               roles => "CREATEDB",
               require => Service["postgresql"] }
```

Figure 1.2 An example IaC script with hard-coded password.

Thesis Statement: Through systematic investigation, we can help practitioners in mitigating quality concerns for infrastructure as code (IaC) scripts by identifying IaC-related anti-patterns that (i) correlate with defects; and (ii) are indicative of security weakness.

In this thesis we make the following contributions:

- Datasets [11] where scripts are labeled as defective (Chapter 3);
- A set of operations that characterize defective IaC scripts (Chapter 5);
- A set of source code properties that relate with defective IaC scripts (Chapter 6);
- Defect prediction models built using code properties and text features of IaC scripts (Chapters 5 and 6);
- A set of development anti-patterns (Chapter 7);
- A set of security anti-patterns for IaC scripts (Chapter 8); and
- A static analysis tool to automatically identify security anti-patterns from IaC scripts (Chapter 8);

We organize rest of the thesis as following: in Chapter 2 we provide necessary background and discuss prior work in the area of IaC. We provide the distribution of defect categories and frequency of defects in Chapter 4. In Chapter 5 we describe the methodology and findings on how we apply text mining and identify operations that characterize defective IaC scripts. We identify source code properties in that are related with defective scripts as described in Chapter 6. In the pursuit of obtaining actionable feedback for practitioners we identify development anti-patterns for IaC scripts in Chapter 7. We focus on security-specific anti-patterns in Chapter 8 and describe our methodology of identifying security anti-patterns in Chapter 8. We discuss our findings in Chapter 9. Finally, we conclude the thesis in Section 10.2.

[11]https://figshare.com/s/c88ece116f803c09beb6

CHAPTER

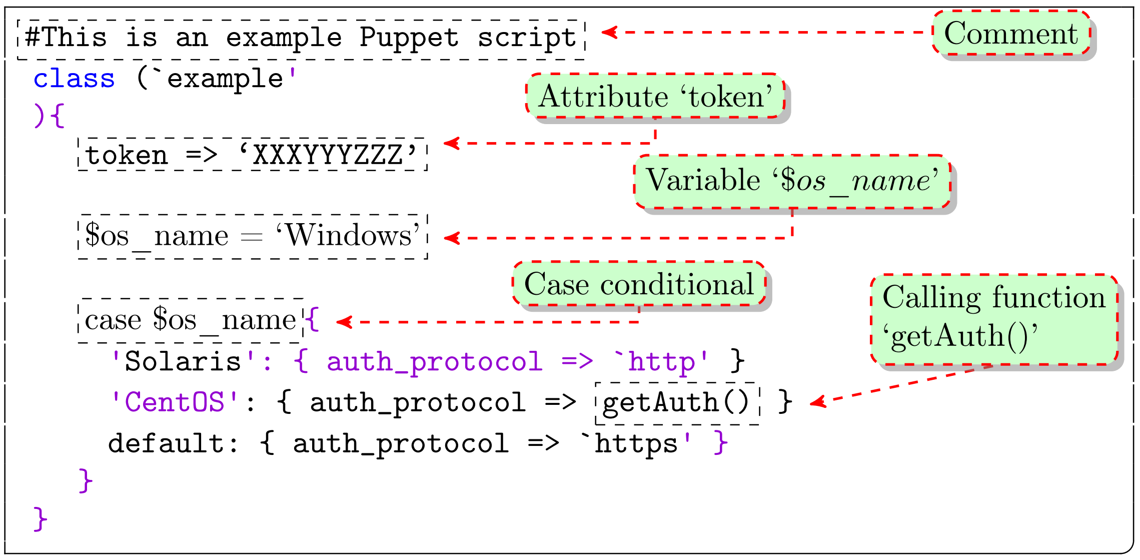
2 BACKGROUND AND RELATED WORK IN THE DOMAIN OF IAC

We provide necessary background and discuss related academic work in this section. First, we provide a brief background on Puppet, a popular tool to implement IaC. Then we provide background information on building prediction models. We end this section by describing related academic research.

2.1 Infrastructure as Code Scripts

Practitioners attribute the concept of infrastructure as code to Chad Fowler, within his blog published in 2013 [1]. IaC scripts use domain specific language (DSL) [Sha16a]. CD organizations widely use commercial systems, such as Puppet, to implement IaC [HF10] [JA15] [Sha16a]. We provide background on Ansible, Chef, and Puppet scripts, as we use Ansible, Chef, and Puppet scripts to evaluate our methodology. Puppet is an IaC system that is used to manage configurations and automate infrastructure [Lab17]. Puppet uses a domain specific language (DSL) [Sha16a]. Typical entities of Puppet include modules, manifests, and resources [Lab17]. A module is a collection of manifests. Puppet users can create their own module or download existing modules from online repositories, such as PuppetForge. A manifest is composed of resources and declarations of resource

[1]https://www.oreilly.com/ideas/an-introduction-to-immutable-infrastructure



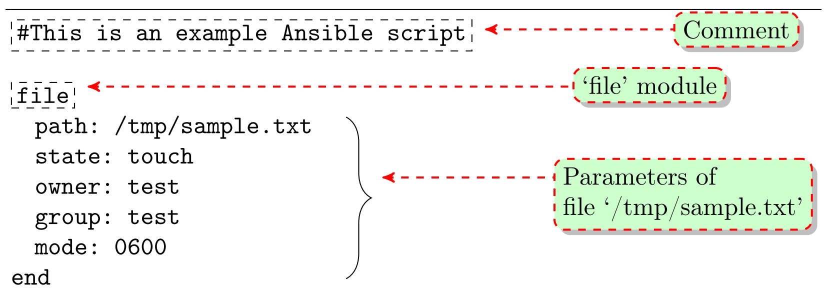
```
#This is an example Puppet script
class (`example'
){
   token => 'XXXYYYZZZ'

   $os_name = 'Windows'

   case $os_name{
     'Solaris': { auth_protocol => `http' }
     'CentOS': { auth_protocol => getAuth() }
     default: { auth_protocol => `https' }
   }
}
```

Figure 2.1 Example of a Puppet script with annotations.

types. Manifests are written as scripts that use a .pp extension. In a single manifest script, multiple declarations of resources are possible. A resource in Puppet has a type, a title, and a mapping of attributes. The resource type determines how the mapping of attributes will be used. For example, a resource of type 'user' can have an attribute called 'uid' to uniquely identify a user. As another example, a resource of type 'file' can have an attribute 'mode' to specify the permissions of the script. In a single manifest script, multiple declarations of resources are possible. Using the 'define' syntax, Puppet users also can create their own resource types with or without using the built-in resources provided by Puppet. Parameters can also be passed to these defined resource types. For better understanding, we provide a sample Puppet code with annotations in Figure 2.1.

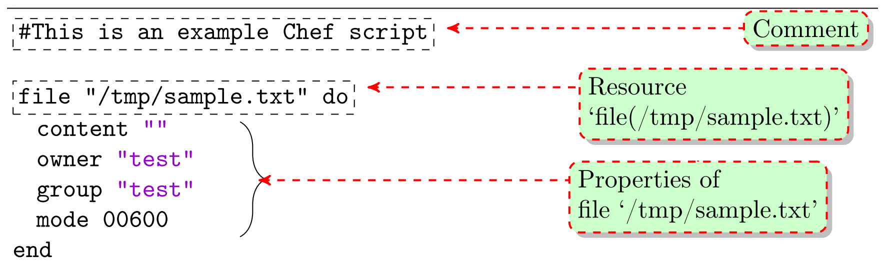
We provide a brief background on Ansible and Chef scripts, which is relevant to conduct our empirical study. Both, Ansible and Chef provided multiple libraries to manage infrastructure and system configurations. In the case of Ansible, developers can manage configurations using 'playbooks', which uses YAML files to manage configurations. For example, as shown in Figure 2.2, an empty file '/tmp/sample.txt' is created using the 'file' module provided by Ansible. The properties of the file such as, path, owner, and group can also be specified. The 'state' property provides options to create an empty file using the 'touch' value, which creates an empty file.

In the case of Chef, configurations as specified using 'recipes', which are domain-specific Ruby scripts. Dedicated libraries are also available to maintain certain configurations. As shown in Figure 2.3, using the 'file' resource, an empty file '/var/sample.txt' is created. The 'content' property is used to specify the content of the file is empty.



```
#This is an example Ansible script

file
  path: /tmp/sample.txt
  state: touch
  owner: test
  group: test
  mode: 0600
end
```

Figure 2.2 Annotation of an example Ansible script.



```
#This is an example Chef script

file "/tmp/sample.txt" do
  content ""
  owner "test"
  group "test"
  mode 00600
end
```

Comment
Resource 'file(/tmp/sample.txt)'
Properties of file '/tmp/sample.txt'

Figure 2.3 Annotation of an example Chef script.

2.2 Prediction Models

In Chapters 5 and 6 we respectively, use text features and source code properties to construct defect prediction models for IaC. In this section we provide necessary background on the technical concepts such as feature selection, parameter tuning of statistical learners, which are used to construct defect prediction models.

2.2.1 Feature Selection

Feature selection is the process of selecting features that have more influence on prediction performance [GE03] [Tan05]. We use the concept of feature selection to identify the metrics that can be effectively used in defect prediction. We use principal component analysis (PCA) [Tan05] for feature selection, because PCA accounts for multi-collinearity amongst features [Tan05], identifies the strongest patterns in the data [Tan05], and has been extensively used in the domain of defect prediction [Nag06] [Gho17] [Nag08]. PCA creates independent linear combinations of the features that account for most of the co-variation of the features. PCA also provides a list of components and the amount of variance explained by each component. These principal components are independent and do not correlate or confound each other. For feature selection, we compute the total amount of variance accounted by the PCA analysis to determine what code metrics should be used for building prediction models. We use the principal components that account for at least 95% of the total variance, as input to statistical learners. We also record the rank (determined by PCA) of each metric to identify which metric has more influence than others.

2.2.2 Statistical Learners

Researchers use statistical learners to build prediction models that learn from historic data and make prediction decisions on unseen data. Previously, researchers used a wide range of statistical learners to build defect prediction models, such as logistic regression and random forest [Les08] [Gho15]. We use three statistical learners that we briefly describe, and reasons for selecting these learners, as following. We use the Scikit Learn API [Ped11] to implement these statistical learners.

- **Classification and Regression Tree (CART)**: CART generates a tree based on the impurity measure, and uses that tree to provide decisions based on input features [Bre84]. We select CART because this learner does not make any assumption on the distribution of features, and is robust to model over-fitting [Tan05] [Bre84].
- **Logistic Regression (LR)**: LR estimates the probability that a data point belongs to a certain class, given the values of features [Fre05]. LR provides good performance for classification if the features are roughly linear [Fre05]. We select LR because this learner performs well for classification problems [Fre05], such as defect prediction [RD13] and fault prediction [Hal12].
- **Naive Bayes (NB)**: The Naive Bayes (NB) classification technique computes the posterior probability of each class to make prediction decisions. We select NB because this learner performs well for classification problems [Fre05], such as defect prediction [RD13] and fault prediction [Hal12].
- **Random Forests (RF)**: RF is an ensemble technique that creates multiple classification trees, each of which are generated by taking random subsets of the training data [Bre01] [Tan05]. Unlike LR, RF does not expect features to be linear for good classification performance. Researchers [Gho15] recommended the use of statistical learners that uses ensemble techniques to build defect prediction models.

Prediction performance measures: We use two measures to evaluate prediction performance of the constructed models:

- **Precision**: Precision measures the proportion of IaC scripts that are actually defective given that the model predicts as defective. We use Equation 2.1 to calculate precision.

$$Precision = \frac{TP}{TP + FP} \tag{2.1}$$

- **Recall**: Recall measures the proportion of defective IaC scripts that are correctly predicted by the prediction model. We use Equation 2.2 to calculate recall.

$$Recall = \frac{TP}{TP + FN} \tag{2.2}$$

- **Area Under the Receiver Operating Characteristic Curve (AUC)**: AUC uses the receiver operating characteristic (ROC). ROC is a two-dimensional curve that plots the true positive rates against false positive rates. An ideal prediction model's ROC curve has an area of 1.0. A random prediction's ROC curve has an area of 0.5. We refer to the *a*rea *u*nder the ROC *c*urve as AUC throughout the paper. We consider AUC as this measure is threshold independent unlike precision and recall [Gho15], and recommended by prior research [Les08].

- **F-Measure**: F-Measure is the harmonic mean of precision and recall. Increase in precision, often decreases recall, and vice-versa [Men07b]. F-Measure provides a composite score of precision and recall, and is high when both precision and recall is high.

- **Accuracy**: We calculate accuracy using Equation 2.3:

$$Accuracy = \frac{TP + TN}{P + N} \tag{2.3}$$

- **G-Mean**: G-mean is the harmonic mean of precision and recall.

2.2.3 Automated Parameter Tuning for Learners

From prior work [Fu16] [Tan16], we observe empirical evidence on how prediction performance can be improved by tuning parameters of statistical learners. Taking motivation from these studies, we tune parameters by executing two tasks: first we identify the parameters of statistical learners needed for tuning. Next, we use an automated technique to tune the identified parameters.

Identifying parameters to tune: To accomplish the first task, we review existing literature in the following manner:

- **Step-1**: We identify the publications that used parameters of statistical learners for defect prediction models. We search the ACM Digital Library, IEEEXplore, ScienceDirect, and Springer using keywords 'parameter tuning', 'defect prediction', and 'software engineering'.

- **Step-2**: Next, we read the collected publication from Step-1 to identify if the publication (i) reported the use of a statistical learner; and (ii) reported which parameters were tuned to report variation in prediction performance for the learner.

- **Step-3**: We read the publications collected from Step-2, to determine if they used CART, LR, or RF. If so, from these publications we determine which parameter of each statistical learner needs to be tuned, and the ranges of values we can select.

Automated technique to tune parameters: We use differential evolution (DE) [SP97], a search-based algorithm, to automatically tune the parameters of the statistical learners. We select DE because using DE as a parameter tuning technique, researchers have observed improved prediction performance for software defects [Fu16]. For a given measure of quality and a set of input parameters that needs to be tuned, DE iteratively identifies the combination of parameter values for which the given measure of quality is optimal. Each combination of parameter value is referred as a 'candidate solution' in DE. DE achieves optimization by generating a population of candidate solutions and creating new candidate solutions by combining existing ones. Four attributes of DE can be set to control the generation of populations: $GENERATION$, $POPULATION$, cross-over probability (CR), and mutation constant (F).

In our case, parameters of each statistical learner is the set of input parameters that need to be tuned. The given measure of quality is the prediction performance measure. We set $GENERATION$, $POPULATION$, CR, and F to respectively, 50, 10, 0.50, and 0.50. We apply DE-based parameter tuning for the evaluation methods used in the paper.

As we use two prediction performance measures, AUC and F-Measure, we repeat the above-mentioned process separately for AUC and F-Measure. When AUC is set as the given measure of quality, as output DE provides the best AUC it was able to achieve. In case of F-Measure, DE provides the best F-Measure it was able to achieve as output.

Comparing prediction performance: To determine if parameter tuning statistically improves performance, we use a variant of the Scott Knott (SK) test [Tan17]. This variant of SK does not assume input to be normal and accounts for negligible effect size [Tan17]. SK uses hierarchical clustering analysis to partition the input data into significantly ($\alpha = 0.05$) distinct ranks [Tan17]. According to SK, a learner for which parameter tuning is applied ranks higher if prediction performance is significantly higher. For example, if tuned CART ranks higher than that of non-tuned CART, then we can state that parameter tuning significantly increases prediction performance. We use SK to compare if parameter tuning significantly increases AUC and F-Measure for all three statistical learners, and for the evaluation methods used.

2.2.4 Evaluation Method

We use 10×10-fold cross validation to evaluate the constructed prediction models. We use the $10times10$-Fold cross validation evaluation approach by randomly partitioning the dataset into 10 equal sized subsamples or folds [Tan05]. The performance of the constructed prediction models are tested by using nine of the 10 folds as training data, and the remaining fold as test data. Similar

Table 2.1 Mapping Between Each Topic and Publication

Topic	**Publication**
Framework/Tool	S6, S10, S12, S13, S15, S18, S19, S20, S21, S22, S24, S25, S26, S29, S30, S31
Adoption of IaC	S1, S3, S7, S9, S14, S16, S17, S18, S23, S27, S28, S32
Empirical Study	S2, S4, S5, S8, S10, S11, S23
Testing	S4, S5, S6, S11

to prior research [Gho15], we repeat the 10-fold cross validation 10 times to assess the statistical learner's prediction stability. We report the median prediction performance score of the 10 runs.

2.3 Related Work in the Domain of IaC

Our thesis is closely related to research studies that have investigated IaC scripts, which we briefly describe below.

Following Petersen et al. [Pet08]'s guidelines, we conduct a systematic mapping study on IaC research in software engineering [Rah18a] starting with 9,387 publications from five scholar databases: ACM Digital Library, IEEE Xplore, Springer, ScienceDirect, and Wiley Online. After applying an inclusion and exclusion criteria, we identify a set of 32 publications which we use to conduct a systematic mapping study. Each of the publications' names are listed in Table A2 of the Appendix. We index each publications as 'S#', for example the index 'S1' refers to the publication 'Cloud WorkBench: Benchmarking IaaS Providers Based on Infrastructure-as-Code'.

We identify the topics that have been researched in the area of IaC by applying qualitative analysis on the content of the 32 publications. Through our qualitative analysis, we identify four topics. A publication can belong to multiple topics implying that the identified topics are not orthogonal to each other. The topics are: (1) Framework/Tool for infrastructure as code (Framework/Tool); (2) Adoption of infrastructure as code (Adoption of IaC); (3) Empirical study related to infrastructure as code (Empirical); and (4) Testing in infrastructure as code (Testing).

A complete mapping between each of the 32 publications and their corresponding topic is available in Table 2.1. We describe each topic, along with the count of publications for each topic as following:

- **Framework/Tool for infrastructure as code (16)**: The most frequently studied topic in IaC-related publications is related to framework or tools. In these publications, authors propose a framework or a tool either to implement the practice of IaC or to extend a functionality of IaC. We describe a few publications related to 'Framework/Tool for IaC' briefly:

  Authors in S12 observed that a wide variety of reusable DevOps artifacts, such as Chef cookbooks and Puppet modules, are shared, but these artifacts are usually bound to specific tools. The authors proposed a novel framework that generates standard Topology and Orchestration Specification for Cloud Applications [2] (TOSCA)-based DevOps artifacts to consolidate DevOps artifacts from different sources. Later, the authors of S12 extend their work in S19, where they constructed a run-time framework using a open source tool-chain to support integration for a variety of DevOps artifacts. In S22, the authors propose a the hidden master framework to assess the survivability of IaC scripts, when tested under simulated attacks. In S24, the authors proposes a tool called ConfigValidator that validates IaC artifacts, such as Docker images, by a writing rules to detect configurations. In S10, the authors propose and evaluate Tortoise, which fixes configurations in Puppet scripts automatically. In S20, 'Charon' is proposed to implement the practice of IaC. We divide the 16 publications into three sub-categories:

– Tools (S12, S13, S15, S18, S19, S20, S21, S24, S25, S26, S29, S30, S31): Publications belonging to this category propose tools to extend a functionality of IaC. Existing tools can be limiting, which may be motivating researchers to propose framework or tools that mitigate these limitations.
– Program repair (S10): Publications belonging to this category are related to repairing of IaC scripts.
– Reliability (S6, S22): Publications belonging to this category are related to the reliability of IaC scripts.

- **Adoption of infrastructure as code (12)**: Publications that relate to this topic discuss how IaC can be used in different domains of software engineering, such as monitoring of system and automated deployment of enterprise applications. We describe the publications that related to this topic briefly as following:

  S1 uses IaC to build a benchmark tool to assess the performance of cloud applications. Authors in S3 and S14 discusses how IaC can be used to implement DevOps. S7 focuses on how Ansible can be used to automatically provision an enterprise application. In S9, authors investigated the feasibility of using Puppet modules to deploy a software-as-a-service (SaaS) application. They observed that Puppet modules are adequate for provisioning SaaS applications, but comes with an extra layer of complexity. In S17, the authors propose 'DevOpsLang' that uses Chef to automatically deploy a chat application. Authors in S18 propose the ABS Modeling Language that uses IaC to deploy an e-commerce application. In S23, authors interview practitioners from 10 companies on the use of IaC in continuous deployment. The authors reported that IaC scripts have fundamentally changed how IT organizations are managing their servers using IaC. They also reported that, similar to the software code base, IaC code bases change frequently. Authors in S27 proposes Omnia that uses IaC to create a monitoring framework to monitor DevOps operations.

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

We further group the 12 publications into following sub-categories:

  - Automated provisioning (S7, S9, S14, S18): Publications belonging to this category are related to automated provisioning of software and services using IaC.
  - Benchmark (S1, S32): Publications belonging to this category are related to benchmark creation of cloud applications.
  - Continuous deployment (S23): Publications belonging to this category are related to use of IaC to implement continuous deployment.
  - DevOps (S3, S17, S27, S28): Publications belonging to this category are related to use of IaC to implement DevOps.
  - Microservices (S16): Publications belonging to this category are related to use of IaC to implement microservices.

  Our analysis suggests that the use of IaC is not only limited to implement automated deployment and DevOps, but also to create monitoring applications for software systems. One possible explanation can be the ability to express system configurations in a programmatic manner using IaC scripts.

- **Empirical study related to infrastructure as code (7)**: We identify seven publications that belong to this category, which apply empirical analysis such as qualitative and quantitative analysis to investigate research questions related to IaC. Of the seven publications that belong to this group, three are focused on testing, and four publications are focused on non-testing issues. The four publications that have conducted empirical analysis, but are not focused on testing are S2, S8, S10, and S23. Publications that have conducted empirical studies related to IaC can be divided into following groups:

  - Testing (S4, S5, S11): Publications belonging to this category are related to testing of IaC scripts.
  - Co-evolution (S2): Publications belonging to this category are related to studying the co-evolution of IaC scripts with other software artifacts, such as Makefiles.
  - Code quality (S8, S10): Publications belonging to this category are related to code quality of IaC scripts.
  - Practitioner Survey (S23): Publications belonging to this category are related to surveying practitioners.

- **Testing in infrastructure as code (4)**: We identify four publications that addresses the topic of testing for IaC scripts. In S6, the authors proposed a testing framework that test if Puppet scripts reach their convergence to the desired system state. S5 proposed a framework to test idempotence in IaC. Their approach used a state transition-based modeling approach to generate test cases to test idempotence for Chef scripts. In S4, the authors reported that the approach suggested in S5 generates too many test cases, and proposed an approach to reduce the amount of test cases to generate the test cases needed for testing of idempotence. The approach proposed in S4 combined testing and static verification approaches to generate test cases needed to test idempotence. We group the four publications into following sub-categories:

– Idempotence (S5): The Publication belonging to this category is related to idempotence. In IaC, it is expected that the deployed system converge into the desired state. Whether or not the deployed system has reached the desired state is called idempotence [Hum13]
– Test Case Reduction (S4): The publication belonging to this category is related to test case reduction.
– Framework (S6, S11): Publications belonging to this category are related to creation of a testing framework.

Based on the above-discussion we identify the following research avenues, which motivated us to conduct the five empirical studies presented in this thesis dissertation:

- **Anti-patterns**: Anti-patterns are recurring practices in software engineering that can have potential negative consequences [Bro98b]. Researchers can explore anti-patterns that may exist for IaC, for example, process anti-patterns, system architecture anti-patterns, security anti-patterns, and project management anti-patterns.

- **Defect Analysis**: Defects in IaC scripts can have serious consequences, for example a defect in an IaC script caused a wide-scale outage for GitHub [3]. Based on our analysis, we do not observe existing IaC-related publications to study defects. We encourage researchers to investigate which characteristics of IaC correlate with defects, and how such defects can be mitigated.

- **Security**: As IaC scripts are used to configure software systems and cloud instances at scale, an error that violates security objectives [Kis11], can compromise the entire system. In our set of 32 publications, we did not find any publication that focus on security issues. Researchers can systematically study which security flaws are exhibited in IaC scripts, what are the consequences of such security flaws, and provide guidelines on how such flaws can be mitigated. One approach to identify security flaws in IaC scripts is to apply qualitative analysis and leverage the weaknesses listed in the Common Weakness Enumeration (CWE) database [4].

In this thesis, we identify operations and source code properties that correlate with defective IaC scripts. We also identify security and development anti-patterns that appear for IaC scripts.

[3]https://github.com/blog/1759-dns-outage-post-mortem
[4]https://cwe.mitre.org/

CHAPTER

3 DEFECT DATASET

In this chapter, we describe the methodology and summary for the defect datasets that we used in Chapters 4, 5, 6, and 7. The defect datasets are not used in Chapter 8.

3.1 Methodology for Dataset Construction

We describe the methodology to construct our datasets as following:

3.1.1 Filtering for Repository Collection

We construct IaC-specific datasets to evaluate our methodology and build prediction models. For our datasets, we use OSS repositories maintained by four organizations: Mirantis [1], Mozilla [2], Openstack [3], and Wikimedia Commons [4]. For Mirantis, Mozilla, Openstack, and Wikimedia we, respectively, collect 26, 1594, 1253, and 1638 repositories. We apply the following selection criteria to construct our datasets:

- **Criteria-1**: The repository must be available for download.

[1]https://github.com/Mirantis
[2]https://hg.mozilla.org/
[3]https://git.openstack.org/cgit
[4]https://gerrit.wikimedia.org/r/#/admin/projects/

Table 3.1 Filtering criteria to construct defect datasets

Criteria	**Dataset**			
	Mirantis	Mozilla	Openstack	Wikimedia
Criteria-1	26	1,594	1,253	1,638
Criteria-2	20	2	61	11
Criteria-3	20	2	61	11
Final	20	2	61	11

- **Criteria-2**: At least 11% of the files belonging to the repository must be IaC scripts. Jiang and Adams [JA15] reported that in OSS repositories, IaC scripts co-exist with other types of files, such as Makefiles and source code files. They observed a median of 11% of the files to be IaC scripts. By using a cutoff of 11%, we assume to collect a set of repositories that contain sufficient amount of IaC scripts for analysis.

- **Criteria-3**: The repository must have at least two commits per month. Munaiah et al. [Mun17] used the threshold of at least two commits per month to determine which repositories have enough development activity for software organizations.

3.1.2 Results of Applying Filtering Criteria for Repository Collection

We apply the three selection criteria presented in Section 3.1.1 to identify the repositories that we use for analysis. We describe how many of the repositories satisfied each of the three criteria in Table 3.1. Each row corresponds to the count of repositories that satisfy each criteria. For example, 26 repositories satisfy Criteria-1, for Mirantis. We obtain 94 repositories to extract Puppet scripts from the four datasets.

3.1.3 Commit Message Processing

Prior research [Ray16] [Zha16] [Zha17] leveraged OSS repositories that use version control systems (VCS) for defect prediction studies. We use two artifacts from the VCS of the selected repositories from Section 3.1.1, to construct our datasets: (i) commits that indicate modification of IaC scripts; and (ii) issue reports that are linked with the commits. We use commits because commits contain information on how and why a file was changed. Commits can also include links to issue reports. We use issue report summaries because they can give us more insights on why IaC scripts were changed in addition to what is found in commit messages. We collect commits and other relevant information in the following manner:

- First, we extract commits that were used to modify at least one IaC script. A commit lists the changes made on one or multiple files [Ala08].

- Second, we extract the message of the commit identified from the previous step. A commit includes a message, commonly referred as a commit message. The commit messages indicate why the changes were made to the corresponding files [Ala08].

- Third, if the commit message included a unique identifier that maps the commit to an issue in the issue tracking system, we extract the identifier and use that identifier to extract the summary of the issue. We use regular expression to extract the issue identifier. We use the corresponding issue tracking API to extract the summary of the issue; and

- Fourth, we combine the commit message with any existing issue summary to construct the message for analysis. We refer to the combined message as 'extended commit message (XCM)' throughout the rest of the dissertation. We use the extracted XCMs to separate the defect-related commits from the non-defect-related commits i.e. commits that are marked not to be defect-related by the rater, and indicative of feature implementation, as described in Section 3.1.4.

3.1.4 Determining Defect-related Commits

We use defect-related commits to identify the defective IaC scripts and the source code properties that characterizes defective IaC scripts. We apply qualitative analysis to determine which commits were defect-related commits. We perform qualitative analysis using the following three steps:

Categorization Phase: At least two raters with software engineering experience determine which of the collected commits are defect-related. We adopt this approach to mitigate the subjectivity introduced by a single rater. Each rater determines an XCM as defect-related if it represents a quality concern in an IaC script. We provide raters with a Puppet documentation guide [Lab17] so that raters can obtain background on Puppet. We also provide the raters the IEEE publication on anomaly classification [IEE10] to help raters to gather background of defect in software engineering. The number of XCMs to which we observe agreements amongst the raters are recorded and the Cohen's Kappa [Coh60] score is computed.

Resolution Phase: Raters can disagree if a commit is defect-related. In these cases, we use an additional rater's opinion to resolve such disagreements. We refer to the additional rater as the 'resolver'.

Practitioner Agreement: To evaluate the ratings of the raters in the categorization and the resolution phase, we randomly select 50 XCMs for each dataset and contact practitioners. We ask the practitioners if they agree to our categorization of XCMs. High agreement between the raters' categorization and programmers' feedback is an indication of how well the raters performed. The percentage of XCMs to which practitioners agreed upon is recorded and the Cohen's Kappa score is computed.

Upon completion of these three steps, we can classify which commits and XCMs are defect-related. We use the defect-related XCMs to identify the source code properties needed to answer the research questions. From the defect-related commits, we determine which IaC scripts are defective, similar to prior work [Zha16]. Defect-related commits list which IaC scripts were changed, and from this list we determine which IaC scripts are defective. A defective script can be included in one or multiple defect-related commits, whereas for neutral scripts are not included in any defect-related commit. The results of applying the presented methodology in this section is presented in Section 3.1.5.

3.1.5 Results of Applying Qualitative Rating for Dataset construction

We construct four datasets by collecting repositories from Mirantis, Mozilla, Openstack, and Wikimedia Commons. We apply the following phases using 89 raters:

- **Categorization Phase**:
  - **Mirantis**: We recruit students in a graduate course related to software engineering via e-mail. The number of students in the class was 58, and 32 students agreed to participate. We follow Internal Review Board protocol (IRB), IRB#12130, in recruitment of students and assignment of defect categorization tasks. We randomly distribute the 1,021 XCMs amongst the students such that each XCM is rated by at least two students. The average professional experience of the 32 students in software engineering is 1.9 years. On average, each student took 2.1 hours.
  - **Mozilla**: One second year PhD student and one fourth year PhD student separately apply qualitative analysis on 3,074 XCMs. The fourth and second year PhD student, respectively, have a professional experience of three and two years in software engineering. The fourth and second year PhD student, respectively, took 37.0 and 51.2 hours to complete the categorization.
  - **Openstack**: One second year PhD student and one first year PhD student separately, apply qualitative analysis on 7,808 XCMs from Openstack repositories. The second and first year PhD student, respectively, have a professional experience of two and one years in software engineering. The second and first year PhD student completed the categorization of the 7,808 XCMs, respectively, in 80.0 and 130.0 hours.
  - **Wikimedia**: 54 graduate students recruited from the 'Software Security' course are the raters. We randomly distribute 972 XCMs amongst the students such that each XCM

is rated by at least two students. According to our distribution, 140 XCMs are assigned to each student. The average professional experience of the 54 students in software engineering is 2.3 years. On average, each student took 2.1 hours to categorize the 140 XCMs. The IRB protocol was IRB#9521.

- **Resolution Phase**:
  - **Mirantis**: Of the 1,021 XCMs, we observe agreement for 509 XCMs and disagreement for 512 XCMs, with a Cohen's Kappa score of 0.21. Based on Cohen's Kappa score, the agreement level is 'fair' [LK77].
  - **Mozilla**: Of the 3,074 XCMs, we observe agreement for 1,308 XCMs and disagreement for 1,766 XCMs, with a Cohen's Kappa score of 0.22. Based on Cohen's Kappa score, the agreement level is 'fair' [LK77].
  - **Openstack**: Of the 7,808 XCMs, we observe agreement for 3,188 XCMs, and disagreements for 4,620 XCMs. The Cohen's Kappa score was 0.21. Based on Cohen's Kappa score, the agreement level is 'fair' [LK77].
  - **Wikimedia**: Of the 972 XCMs, we observe agreement for 415 XCMs, and disagreements for 557 XCMs, with a Cohen's Kappa score of 0.23. Based on Cohen's Kappa score, the agreement level is 'fair' [LK77].

The author of this dissertation was the resolver and resolved disagreements for all four datasets. In case of disagreements, the resolver's categorization is considered as final.

We observe that the raters' agreement level to be 'fair' for all four datasets. One possible explanation can be that the raters agreed on whether an XCM is defect-related but disagreed on which category of the defect is related to. For defect categorization, fair or poor agreement amongst raters however, is not uncommon. Henningsson et al. [HW04] also reported a low agreement amongst raters.

Practitioner Agreement: We report the agreement level between the raters' and the practitioners' categorization for randomly-selected 50 XCMs as following:

- **Mirantis**: We contact three practitioners and all of them respond. We observe an 89.0% agreement with a Cohen's Kappa score of 0.8. Based on Cohen's Kappa score, the agreement level is 'substantial' [LK77].
- **Mozilla**: We contact six practitioners and all of them respond. We observe a 94.0% agreement with a Cohen's Kappa score of 0.9. Based on Cohen's Kappa score, the agreement level is 'almost perfect' [LK77].

– **Openstack**: We contact 10 practitioners and all of them respond. We observe a 92.0% agreement with a Cohen's Kappa score of 0.8. Based on Cohen's Kappa score, the agreement level is 'substantial' [LK77].

– **Wikimedia**: We contact seven practitioners and all of them respond. We observe a 98.0% agreement with a Cohen's Kappa score of 0.9. Based on Cohen's Kappa score, the agreement level is 'almost perfect' [LK77].

We observe that the agreement between ours and the practitioners' categorization varies from 0.8 to 0.9, which is higher than that of the agreement between the raters in the Categorization Phase. One possible explanation can be related to how the resolver resolved the disagreements. The author of this dissertation has industry experience in writing IaC scripts, which may help to determine categorizations that are consistent with practitioners. Another possible explanation can be related to the sample provided to the practitioners. The provided sample, even though randomly selected, may include commit messages whose categorization are relatively easy to agree upon.

3.2 Summary of Constructed Datasets

We construct datasets using Puppet scripts from OSS repositories maintained by four organizations: Mirantis, Mozilla, Openstack, and Wikimedia Commons. We select Puppet because it is considered as one of the most popular tools to implement IaC [JA15] [Sha16a], and has been used by organizations since 2005 [MJ11]. Mirantis is an organization that focuses on the development and support of cloud services such as OpenStack [5]. Mozilla is an OSS community that develops, uses, and supports Mozilla products such as Mozilla Firefox [6]. Openstack foundation is an OSS platform for cloud computing where virtual servers and other resources are made available to customers [7]. Wikimedia Foundation is a non-profit organization that develops and distributes free educational content [8].

We report summary statistics on the collected repositories in Table 3.2. According to Table 3.2, for Mirantis we collect 180 Puppet scripts that map to 1,021 commits. The constructed datasets used for empirical analysis are available online [9].

[5]https://www.mirantis.com/

[6]https://www.mozilla.org/en-US/

[7]https://www.openstack.org/

[8]https://wikimediafoundation.org/

[9]https://figshare.com/s/c88ece116f803c09beb6

Table 3.2 Statistics of Four Datasets

Attributes	Mirantis	Mozilla	Openstack	Wikimedia
Puppet Scripts	180	299	1,363	296
Commits with Puppet Scripts	1,021	3,074	7,808	972
Commits with Report IDs	82 of 1021, 8.0%	2764 of 3074, 89.9%	2252 of 7808, 28.8%	210 of 972, 21.6%
Defect-related Commits	344 of 1021, 33.7%	558 of 3074, 18.1%	1987 of 7808, 25.4%	298 of 972, 30.6%
Defective Puppet Scripts	96 of 180, 53.3%	137 of 299, 45.8%	793 of 1363, 58.2%	160 of 296, 54.0%

CHAPTER

4 DEFECT CATEGORIES AND FREQUENCIES

Categorization of defects can guide organizations on how to improve their development process. Chillarege et al. [Chi92] proposed the orthogonal defect classification (ODC) technique which included a set of defect categories. According to Chillarege et al. [Chi92], each of these defect categories map to a certain activity of the development process which can be improved. Since the introduction of ODC in 1992, companies such as IBM [But02], Cisco [1], and Comverse [LC03] have successfully used ODC to categorize defects. Such categorization of defects help practitioners to identify process improvement opportunities for software development. For example, upon adoption of ODC practitioners from IBM [But02] reported "*The teams have been able to look at their own data objectively and quickly identify actions to improve their processes and ultimately their product. The actions taken have been reasonable, not requiring huge investments in time, money, or effort*". A systematic categorization of defects in IaC scripts can help in understanding the nature of IaC defects and in providing actionable recommendations for practitioners to mitigate defects in IaC scripts. In this chapter, we focus on the categorization of defects found in IaC scripts [Rah18b]. We organize this chapter by first providing related work. Next, we provide methodology. Finally, we provide results of the study.

[1]https://www.stickyminds.com/sites/default/files/presentation/file/2013/t12.pdf

4.1 Related Work in Non-IaC Domain

Categorization of defects for a software system helps in formulating effective mitigation strategies, in improving the functionality provided by the software system [PR12] [Thu12], and in prioritizing testing efforts [Chi92] [Cot13]. Researchers have previously used classification schemes, such as the orthogonal defect classification (ODC) [Chi92], to classify defects for non-IaC software systems, such as, air flight systems [Lyu03], database systems [PR12], operating systems [Cot13], written in GPLs. Duraes and Madeira [DM06] studied 668 faults from 12 software systems and reported that 43.4% of the 668 defects were algorithm defects, and 21.9% of the defects were assignment-related defects by applying ODC. Fonseca et al. [FV08] used ODC to categorize security defects that appear in web applications. They collected 655 security defects from six PHP web applications and reported 85.3% of the security defects belong to the algorithm category. Zheng et al. [Zhe06a] applied ODC on telecom-based software systems and observed an average of 35.4% of defects belong to the algorithm category. Lutz and Mikluski [LM04] studied defect reports from seven missions of NASA and observed functional defects to be the most frequent category of 199 reported defects. Christmasson and Chillarege [CC96] studied 408 IBM OS faults extracted and reported 37.7% and 19.1% of these faults to belong to the algorithm and assignment categories, respectively. Basso et al. [Bas09] studied defects from six Java-based software namely, Azureus, FreeMind, Jedit, Phex, Struts, and Tomcat, and observed the most frequent category to be algorithm defects. Cinque et al. [Cin14] analyzed logs from an industrial system that belong to the air traffic control system and reported that 58.9% of the 3,159 defects were classified as algorithm defects.

Unlike non-IaC software systems, IaC systems use DSLs [Sha16a]. DSLs are fundamentally different from GPLs with respect to comprehension, semantics, and syntax [Hud98] [Voe13]. We hypothesize that the fundamental differences between IaC systems and non-IaC systems can lead to a different defect category distribution for IaC systems. Systematic investigation of defects in IaC systems can help in understanding the nature of IaC defects, and in providing evidence that supports or negates our hypothesis.

We answer the following research questions:

- RQ1: How frequently do defects occur in infrastructure as code scripts?
- RQ2: What is the distribution of defect category occurrences in infrastructure as code scripts, using orthogonal defect classification (ODC)?
- RQ3: How do defect categories relate with size of infrastructure as code (IaC) scripts?
- RQ4: How does the distribution of defect categories in infrastructure as code (IaC) scripts compare with other non-IaC software systems, as published in the literature?

Table 4.1 Criteria for Determining Categories

Category	**Criterion**
Algorithm (AL)	Indicates efficiency or correctness problems that affect task and can be fixed by re-implementing an algorithm or local data structure.
Assignment (AS)	Indicates changes in a few lines of code.
Build/Package/Merge (B)	Indicates defects due to mistakes in change management, library systems, or version control systems.
Checking (C)	Indicates defects related to data validation and value checking.
Documentation (D)	Indicates defects that affect publications and maintenance notes.
Function (F)	Indicates defects that affect significant capabilities.
Interface (I)	Indicates defects in interacting with other components, modules, or control blocks.
No Defect (N)	Indicates no defects.
Other (O)	Indicates a defect that do not belong to the categories: AL, AS, B, C, D, F, I, N, and T.
Timing/Serialization (T)	Indicates errors that involve real time resources and shared resources.

4.2 Methodology

We select ODC as ODC uses semantic information collected from the software system, and can be used to make informed decisions on the defect categories [Chi92]. According to ODC, a defect can belong to one of the eight categories: algorithm (AL), assignment (AS), build/package/merge (B), checking (C), documentation (D), function (F), interface (I), and timing/serialization (T).

As mentioned in Chapter 3, the collected XCMs derived from commits and issue report descriptions might correspond to feature enhancement or performing maintenance tasks, which are not related to defects. As an XCM might not correspond to a defect, we added a 'No defect (N)' category. Furthermore, a XCM might not to belong to any of the eight categories that belong to ODC. Hence, we introduced the 'Other (O)' category. We classified the XCMs into one of these 10 categories. The criteria for each of the 10 categories are described in Table 4.1.

We performed qualitative analysis on the collected XCMs to determine the category to which a commit belongs. For performing qualitative analysis, we had raters with software engineering experience apply the ODC defect categories on the collected XCMs. We conducted the qualitative analysis in the following manner:

- **Categorization Phase**: We distribute the XCMs in a manner so that each XCM is reviewed by at least two raters. We adopt this approach to mitigate the subjectivity introduced by a single rater. Each rater determines an XCM as defect-related if the XCM represents a quality issue in an IaC script. We provide raters with an electronic handbook on IaC [Lab17], and the ODC publication [Chi92]. We do not provide any time constrain for categorizing the defects. We record the agreement level amongst raters in two ways: i. by counting the XCMs for which the raters had the same rating; and ii. by computing the Cohen's Kappa score [Coh60].

- **Resolution Phase**: Raters can disagree on the category of a commit. In these cases, we use an additional rater's opinion to resolve such disagreements. We refer to the additional rater as the 'resolver'.
- **Practitioner Agreement**: To evaluate the ratings of the raters in the categorization and the resolution phase, we randomly select 50 XCMs for each dataset, and contact practitioners. We ask the practitioners if they agree to our categorization of XCMs. High agreement between the raters' categorization and programmers' feedback is an indication of how well the raters performed. The percentage of XCMs to which practitioners agreed upon should be recorded and the Cohen's Kappa score should be computed.

4.2.1 RQ1: How frequently do defects occur in infrastructure as code scripts?

We use this RQ to assess how frequently defects occur in IaC scripts. We calculate defect density (DD) to answer this research question. We calculate DD in two ways: the count of defects that appear at every line of IaC script (DD_{LOC}), and the count of defects that appear at every 1000 lines of IaC script, similar to prior work [Bat01] [Moh04] [MW03] (DD_{KLOC}). We use Equations 4.1 and 4.2 respectively, to calculate DD_{LOC} and DD_{KLOC}.

$$DD_{LOC} = \frac{\text{total count of defects that appear for all IaC scripts}}{\text{total count of lines for all IaC scripts}} \tag{4.1}$$

$$DD_{KLOC} = \frac{\text{total count of defects that appear for all IaC scripts}}{\frac{\text{total count of lines for all IaC scripts}}{1000}} \tag{4.2}$$

DD_{LOC} and DD_{KLOC} provide us the defect density of the three IaC systems, and gives an assessment of how frequently defects occur in IaC scripts.

4.2.2 RQ2: What is the distribution of defect category occurrences in infrastructure as code scripts, using orthogonal defect classification (ODC)?

We answer this RQ using the categorization achieved through qualitative analysis and by reporting the following measure: *Count of the defect-related commits belong to a certain defect category*: we calculate DCC using Equation 4.3.

$$\begin{gathered}\text{Defect Commits for Category x } (DCC\_x) = \\ \frac{\text{total count of defect-related commits that belong to category } x}{\text{total count of defect-related commits}} * 100\end{gathered} \tag{4.3}$$

4.2.3 RQ3: How do defect categories relate with size of infrastructure as code scripts?

Researchers in prior work [MP93] [FO00] [Hat97] have observed the relationships of source code size and defects for software systems written in GPLs. Similar investigation for IaC scripts can provide valuable insights, and may help practitioners in identifying scripts that contain defects or specific categories of defects. We compare how size of IaC scripts vary from category of defects to another by calculating LOC of scripts that belong to each of the nine defect categories namely, Algorithm (AL), Assignment (AS), Build/Package/Merge (B), Checking (C), Documentation (D), Function (F), Interface (I), Other (O), and Timing/Serialization (T). We apply the Scott-Knott test (SK) to compare if the categories of defects significantly vary from each other with respect to size. SK uses hierarchical clustering analysis to partition the input data into significantly ($\alpha = 0.05$) distinct ranks [Tan17]. According to SK, a learner for which parameter tuning is applied ranks higher if prediction performance is significantly higher. As a hypothetical example, if defect category 'AS' ranks higher than that of 'AL', then we can state that IaC scripts that have 'AS'-related defects are significantly larger than scripts with 'AL'-related defects. A script can belong to multiple defect categories.

This investigation will help us to determine if size significantly varies across the nine defect categories. Along with reporting the SK ranks, we also present the distribution of script sizes in the form of boxplots.

4.2.4 RQ4: How does the distribution of defect categories in infrastructure as code (IaC) scripts compare with other non-IaC software systems, as published in the literature?

We answer this RQ by identifying prior research that have used ODC to categorize defects in other systems, such as safety critical systems [LM04], operating systems [CC96], and Java-based systems [Bas09]. We collected a set of publications based on the following selection criteria:

- **Step-1**: The publication must cite Chillarege et al. [Chi92]'s ODC publication, indexed by IEEE Xplore, and published on or after 2000. By selecting publications on or after 2000, we assume we will get defect categories of software systems that are relevant and comparable against modern software systems, such as Puppet.

- **Step-2**: The publication must use ODC in its original form to categorize defects of a software system. A publication might cite Chillarege et al. [Chi92]'s paper as related work, but not use ODC for categorization. A publication might also modify ODC to form more defect categories and use the modified version of ODC to categorize defects. As we use the original eight categories of ODC, we do not include publications that modified or extended ODC to categorize defects.

- **Step-3**: The publication must explicitly report the systems they studied with a distribution of defects across the ODC defect type categories, along with the total count of bugs, defects, or faults for each software system. Along with defects, we consider bugs and faults, as in prior work researchers have used bugs [Thu12] and faults interchangeably with defects [PR12].

4.3 Results

We categorize XCMs to classify which collected commits are defect-related, using the following phases:

- **Categorization Phase**:
  - **Mozilla**: Two graduate students, separately, apply qualitative analysis on 3,074 XCMs. The first and second rater, respectively, have a professional experience of three and two years in software engineering.
  - **Openstack**: Two graduate students, separately, apply qualitative analysis on 7,808 XCMs from Openstack repositories. The first and second rater, respectively, have a professional experience of two and one years in software engineering.
  - **Wikimedia**: We recruit students in a graduate course related to software engineering titled 'Software Security', via e-mail. The number of students in the class was 74, and 54 students agreed to participate. We follow Institutional Review Board (IRB) protocol (IRB#9521) in recruitment of students and assignment of defect categorization tasks. We randomly distribute the 972 XCMs amongst the students such that each XCM is rated by at least two students. The average professional experience of the 54 students in software engineering is 2.3 years.
- **Resolution Phase**:
  - **Mozilla**: Of the 3,074 XCMs, we observe agreement for 1,308 XCMs and disagreement for 1,766 XCMs, with a Cohen's Kappa score of 0.22. Based on Cohen's Kappa score, the agreement level is 'fair' [LK77].
  - **Openstack**: Of the 7,808 XCMs, we observe agreement for 3,188 XCMs, and disagreements for 4,620 XCMs. The Cohen's Kappa score was 0.21. Based on Cohen's Kappa score, the agreement level is 'fair' [LK77].
  - **Wikimedia**: Of the 972 XCMs, we observe agreement for 415 XCMs, and disagreements for 557 XCMs, with a Cohen's Kappa score of 0.23. Based on Cohen's Kappa score, the agreement level is 'fair' [LK77].

Table 4.2 Defect Density of the Three Infrastructure as Code (IaC) Systems

Metric	Mozilla	Openstack	Wikimedia
DD_{LOC}	0.0184	0.0162	0.0171
DD_{KLOC}	18.4	16.3	17.1

We observe that the raters agreement level to be 'fair' for all three datasets. One possible explanation can be that the raters agreed if a XCM is defect-related, but disagreed on what the category is. Fair or poor agreement amongst raters however, is not uncommon. Henningsson et al. [HW04] reported a low agreement amongst raters for defect classification.

The author of the dissertation is the resolver, and resolve disagreements for the three datasets. We report the agreement level between the raters' and the practitioners' categorization for randomly selected 50 XCMs as following:

– **Mozilla**: We contact six programmers and all of them responded. We observe a 94.0% agreement with a Cohen's Kappa score of 0.9. Based on Cohen's Kappa score, the agreement level is 'almost perfect' [LK77].

– **Openstack**: We contact 10 programmers and all of them responded. We observe a 92.0% agreement with a Cohen's Kappa score of 0.8. Based on Cohen's Kappa score, the agreement level is 'substantial' [LK77].

– **Wikimedia**: We contact seven programmers and all of them responded. We observe a 98.0% agreement with a Cohen's Kappa score of 0.9. Based on Cohen's Kappa score, the agreement level is 'almost perfect' [LK77].

We present the count of defect-related commits, and defective IaC scripts in Table 3.2. We observe for Mozilla, 18.1% of the commits are defect-related, even though 89.9% of the commits included identifiers to issues. According to our qualitative analysis, for Mozilla, issue reports exist that are not related to defects such as, support of new features [2], and installation issues [3]. In case of Openstack and Wikimedia, respectively, 28.8% and 16.4% of the Puppet-related commits include identifiers that map to issues.

4.3.1 RQ1: How frequently do defects occur in infrastructure as code scripts?

We use defect density to answer this research question and report the defect density of the three IaC systems using DD_{LOC} and DD_{KLOC} in Table 4.2.

[2]https://bugzilla.mozilla.org/show_bug.cgi?id=868974
[3]https://bugzilla.mozilla.org/show_bug.cgi?id=773931

Researchers have previously observed that similar to software source code, IaC scripts are susceptible to defects [JA15]. Our findings provide empirical evidence to these observations. The reported defect densities in Table 4.2 have similarities with other software systems as reported in prior research. Hatton [Hat97] have reported in Ada language-based systems the defect density is 6.0 per KLOC. Battin et al. [Bat01] have reported that for CC++-based systems defect density can vary from 0.1 to 1.3 per KLOC. Mohagheghi et al. [Moh04] have reported defect density to vary from 0.7 to 3.7 per KLOC for a telecom software system written in C, Erlang, and Java. For a Java-based system researchers [MW03] have reported a defect density of 4.0 defects/KLOC. To summarize, prior research shows that defect densities can vary from one software system to another, and one organization to another. We observe similar trends for IaC systems as well.

From Table 4.2, we observe that the defect density per KLOC is highest for Mozilla and lowest for Openstack. On average, as shown in Chapter 3.2 Mozilla's IaC scripts are smaller in size, yet the defect occurrences are higher than average. In prior research [Hat97] [FN99] [BP84], researchers have observed that contrary to practitioner perception, with respect to density, smaller software artifacts contain more defects than larger artifacts, and our findings provide further evidence to such observations.

4.3.2 RQ2: What is the distribution of defect category occurrences in infrastructure as code scripts, using orthogonal defect classification (ODC)?

We answer this RQ by first presenting the values for defect-related commit count (DCC) that belong to each defect category Algorithm (AL), Assignment (AS), Build/Package/Merge (B), Checking (C), Documentation (D), Function (F), Interface (I), Other (O), and Timing/Serialization (T). In Figure 4.3 we report the DCC values for the three datasets.

Table 4.3 present the percentage of defect-related commit messages using DCC. According to Table 4.3, assignment-related defects account for 49.3%, 36.5%, 57.6%, and 54.5% of the defect-related commits, respectively for Mirantis, Mozilla, Openstack, and Wikimedia. Together, assignment and checking-related defects account for 53.5%, 64.2%, and 69.2% of the defect messages, respectively for Mozilla, Openstack, and Wikimedia. We observe the dominant defect category for IaC scripts to be 'assignment', which includes defects related to syntax and configuration errors. Accordingly, the ODC process improvement guidelines recommend the teams allocate more code inspection, static analysis, and unit testing effort for IaC scripts. We also observe defects categorized as assignment to be more prevalent amongst IaC scripts compared to the 26 non-IaC software systems.

From Table 4.3, the category Other (O), DCC is 9.4%, 6.7%, and 0.8%, respectively, for Mozilla, Openstack, and Wikimedia. This category of defects includes defect-related commits that correspond to a defect but the rater was not able to identify the category of the defect. Example of such defect-related commit messages include 'minor puppetagain fixes', and 'summary fix hg on osx; a=bustage'.

Category	MIR	MOZ	OST	WIK
Algorithm	6.4	7.7	5.9	3.3
Assignment	49.3	36.5	57.5	62.7
Build/Package/Merge	6.6	6.4	8.6	4.7
Checking	6.7	17.0	6.7	12.1
Documentation	1.9	2.3	2.6	4.3
Functional	7.5	10.0	2.4	4.3
Interface	6.4	1.9	2.9	2.7
Other	12.5	9.5	6.7	0.7
Timing	2.5	8.4	6.5	5.0

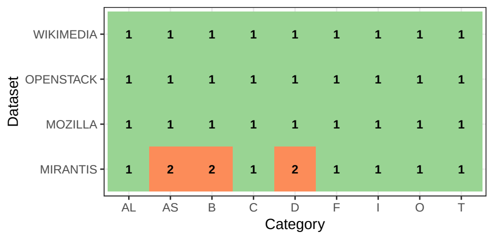
Table 4.3 Distribution of defect categories.

One possible explanation on the categorization of other-related commit messages can be attributed to the information content provided in the messages which might convey information about the presence of a defect, but not enough information to determine what the defect category is. We list an example XCM for each defect category in Table 4.4.

To summarize, we observe assignment-related defects (AS) to be the dominant defect category for all three datasets, followed by checking-related defects (C). One possible explanation can be how IaC scripts are used. IaC scripts are used to manage configurations and deployment infrastructure automatically [HF10]. Examples of such use includes practitioners use IaC to provision cloud instance, such as Amazon Web Services (AWS) [Cit15a], or to manage dependencies of software [HF10]. When assigning configurations of library dependencies or provisioning cloud instances, programmers might be introducing defects that need fixing. Fixing these defects involve few lines of code in IaC scripts, and thus these defects fall in the assignment category. Examples of defect-related XCMs that belong to the assignment category are: 'fix gdb version to be more precise' and 'bug 867593 use correct regex syntax'. Our findings suggest that the identified defect category distribution depends on how IaC systems are used. Our explanation is congruent with prior research: in case of machine learning systems such as Apache Mahout [4], Apache Lucene [5], and OpenNLP [6], that are dedicated for executing machine learning algorithms, researchers have observed the most frequently occurring defects are algorithm-related [Thu12]. For database systems, Sullivan and Chillarege [SC91] have observed that defect distribution is dominated by assignment and checking-related defects. Other researchers [Chi92] find this observation 'reasonable', as few lines of code for database systems typically miss a condition, or assign a wrong value.

[4]http://mahout.apache.org/
[5]https://lucene.apache.org/core/
[6]https://opennlp.apache.org/

4.3.3 RQ3: How do defect categories relate with size of infrastructure as code scripts?



Dataset	AL	AS	B	C	D	F	I	O	T
WIKIMEDIA	1	1	1	1	1	1	1	1	1
OPENSTACK	1	1	1	1	1	1	1	1	1
MOZILLA	1	1	1	1	1	1	1	1	1
MIRANTIS	1	2	2	1	2	1	1	1	1

Figure 4.1 The Scott-Knott (SK) ranks for each defect category and each dataset. Except for Mirantis, the same rank across all defect categories states that with respect to size, there is no significant difference between the defect categories.

In Figure 4.1, we report the results of our SK tests. Except for Mirantis, as shown in Figure 4.1, we do not observe any significant difference between the defect categories, as for all datasets, the rank is same for all nine defect categories. Overall, our findings indicate that defect categories have no relationship with size of scripts.

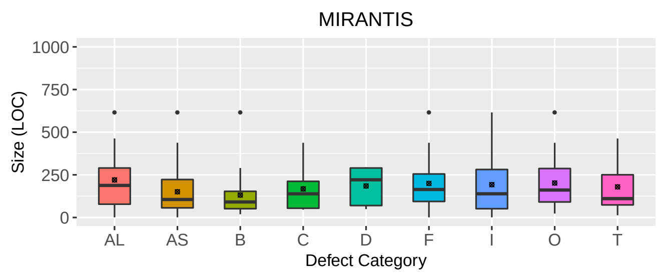
We also present the distribution of size for each defect category in Figure 4.2. The y-axis in each subplot presents the size in LOC for each defect category. For Mozilla, considering average script size, category 'Interface (I)' is the largest, and category 'Checking (C)' is the smallest. For Openstack and Wikimedia, categories 'Documentation (D)' and 'Build/Package/Merge (B)' is the largest, and categories 'Function (F)' and 'Other (O)' is the smallest. Based on average values, we can state that scripts with documentation and build/package/merge-related defects are larger but the difference is not statistically significant.

On possible explanation of our findings can be related to how organizations are developing IaC scripts. For Mozilla, programmers may be inadvertently introducing interface defects as they try to interface with large components. For Openstack, programmers may be introducing documentation-related defects in scripts which have a lot of comments. In future studies, we hope to investigate these findings in details.

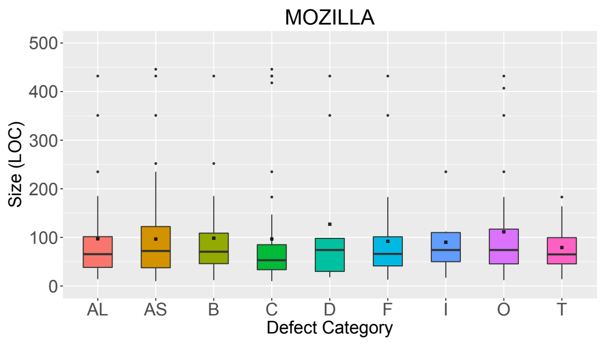
4.3.4 RQ4: How does the distribution of defect categories in infrastructure as code (IaC) scripts compare with other non-IaC software systems, as published in the literature?

We identify 23 software systems using the three steps outlined in Section 4.2.4:

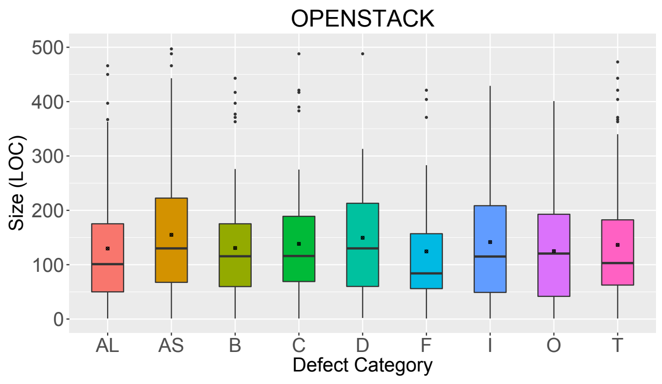
- **Step-1**: As of March 11, 2017, 295 publications indexed by IEEE Xplore, cited Chillarege et al.'s ODC publication [Chi92]. Of the 295 publications 249 publications were published on or after



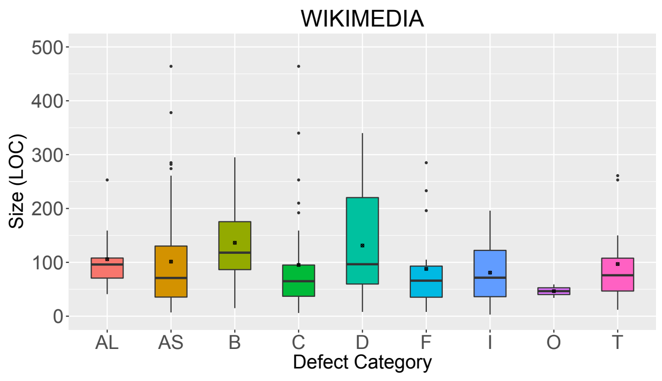
(a)



(b)



(c)



(d)

Figure 4.2 Distribution of script size for each defect category.

2000.

- **Step-2**: Of these 249 publications, 15 applied ODC in its original form to classify defects for software systems.
- **Step-3**: Of these 15 publications, six publications explicitly mentioned the total count of defects and provided a distribution of defect categories.

In Table 4.5, we present the categorization of defects for these 23 software systems. The 'System' column reports the studied software system followed by the publication reference in which the findings were reported. The 'Count' column reports the total count of defects that were studied for the software system. The next eight consecutive columns, respectively, present the eight defect categories from ODC: algorithm (AL), assignment (AS), block (B), checking (C), documentation (D), function (F), interface (I), and timing (T). The 'Lang.' column presents the programming language using which the system is developed.

From Table 4.5, we observe that for four of the 23 software systems, 40% or more of the defect categories belonged to category assignment. For 13 of the 21 software systems, we observed at least 40% of the defects are algorithm-related. We also observe block, documentation, and timing-related defects to rarely occur in previously studied software systems. Considering 23 software systems, on average 38.0% and 27.1% of the defects respectively belong to categories algorithm and assignment. In contrast to IaC systems, assignment-related defects were not prevalent: assignment-related defects were the dominant category for two of the 23 previously studied non-IaC software systems. Our findings support our hypothesis: due to fundamental differences between DSLs and GPLs, IaC systems have a different defect category distribution than non-IaC system written in GPLs.

Summary: In summary, we observe the dominant defect category for IaC scripts to be 'assignment', which includes defects related to syntax and configuration errors. Accordingly, the ODC process improvement guidelines recommend the teams allocate more code inspection, static analysis, and unit testing effort for IaC scripts. We also observe defects categorized as assignment to be more prevalent amongst IaC scripts compared to the 26 non-IaC software systems.

Table 4.4 Examples of Extended Commit Messages (XCMs) for Defect Categories

Category	Mozilla	Openstack	Wikimedia
AL	bug 869897: make watch_-devices.sh logs no longer infinitely growing; my thought is logrotate.d but open to other choices here	fix middleware order of proxy pipeline and add missing modules this patch fixes the order of the middlewares defined in the swift proxy server pipeline	nginx service should be stopped and disabled when nginx is absent
AS	bug 867593 use correct regex syntax	resolved syntax error in collection	fix missing slash in puppet file url
B	bug 774638-concat::setup should depend on diffutils	fix db_sync dependencies: this patch adds dependencies between the cinder-api and cinder-backup services to ensure that db_-sync is run before the services are up. change-id: i7005	fix varnish apt dependencies these are required for the build it does. also; remove unnecessary package requires that are covered by require_package
C	bug 1118354: ensure deploystudio user uid is >500	fix check on threads	fix hadoop-hdfs-zkfc-init exec unless condition $zookeeper_hosts_string was improperly set; since it was created using a non-existent local var '@zoookeeper_hosts'
D	bug 1253309 - followup fix to review comments	fix up doc string for workers variable change-id:ie886	fix hadoop.pp documentation default
F	bug 1292523-puppet fails to set root password on buildduty-tools server	make class rally work class rally is created initially by tool	fix ve restbase reverse proxy config move the restbase domain and 'v1' url routing bits into the apache config rather than the ve config.
I	bug 859799-puppetagain buildbot masters won't reconfig because of missing sftp subsystem	update all missing parameters in all manifests	fix file location for interfaces
N	merge bug 1178324 from default	add example for cobbler	new packages for the server
O	bug 856017 fix puppet_-server_reports	fix api.pp with rabbitmq	fix nginx configpackageservice ordering
T	bug 838203-test_-alerts.html times out on ubuntu 12.04 vm	fix minimal available memory check change-id:iaad0	fix hhvm library usage race condition ensure that the hhvm lib 'current' symlink is created before setting usrbinphp to point to usrbinhhvm instead of usrbinphp5. previously there was a potential for race conditions due to resource ordering rules

Table 4.5 Defect Categories of Previously Studied Software Systems and IaC Systems

System	**Lang.**	**Count**	**AL(%)**	**AS(%)**	**B(%)**	**C(%)**	**D(%)**	**F(%)**	**I(%)**	**T(%)**
Bourne Again Shell (BASH) [Cot13]	C	2.0	0.0	**100.0**	0.0	0.0	0.0	0.0	0.0	0.0
ZSNES-Emulator for x86 [Cot13]	C, C++	3.0	33.3	**66.7**	0.0	0.0	0.0	0.0	0.0	0.0
Pdftohtml-Pdf to html converter [Cot13]	Java	20.0	40.0	**55.0**	0.0	5.0	0.0	0.0	0.0	0.0
Firebird-Relational DBMS [Cot13]	C++	2.0	0.0	**50.0**	0.0	50.0	0.0	0.0	0.0	0.0
Air flight application [Lyu03]	C	426.0	19.0	31.9	0.0	14.0	0.0	**33.8**	1.1	0.0
Apache web server [PR12]	C	1101.0	**47.6**	26.4	0.0	12.8	0.0	0.0	12.9	0.0
Joe-Tex editor [Cot13]	C	78.0	15.3	25.6	0.0	**44.8**	0.0	0.0	14.1	0.0
Middleware system for air traffic control [Cin14]	C	3159.0	**58.9**	24.5	0.0	1.7	0.0	0.0	14.8	0.0
ScummVM-Interpreter for adventure engines [Cot13]	C++	74.0	**56.7**	24.3	0.0	8.1	0.0	6.7	4.0	0.0
Linux kernel [Cot13]	C	93.0	**33.3**	22.5	0.0	25.8	0.0	12.9	5.3	0.0
Vim-Linux editor [Cot13]	C	249.0	**44.5**	21.2	0.0	22.4	0.0	5.2	6.4	0.0
MySQL DBMS [PR12]	C, C++	15102.0	**52.9**	20.5	0.0	15.3	0.0	0.0	11.2	0.0
CDEX-CD digital audio data extractor [Cot13]	C, C++, Python	11.0	9.0	18.1	0.0	18.1	0.0	0.0	**54.5**	0.0
Struts [Bas09]	Java	99.0	**48.4**	18.1	0.0	9.0	0.0	4.0	20.2	0.0
Safety critical system for NASA spacecraft	Java	199.0	17.0	15.5	2.0	0.0	2.5	29.1	5.0	10.5
Azureus [Bas09]	Java	125.0	**36.8**	15.2	0.0	11.2	0.0	28.8	8.0	0.0
Phex [Bas09]	Java	20.0	**60.0**	15.0	0.0	5.0	0.0	10.0	10.0	0.0
TAO Open DDS [PR12]	Java	1184.0	**61.4**	14.4	0.0	11.7	0.0	0.0	12.4	0.0
JEdit [Bas09]	Java	71.0	**36.6**	14.0	0.0	11.2	0.0	25.3	12.6	0.0
Tomcat [Bas09]	Java	169.0	**57.9**	12.4	0.0	13.6	0.0	2.3	13.6	0.0
FreeCvi-Strategy game [Cot13]	Java	53.0	**52.8**	11.3	0.0	13.2	0.0	15.0	7.5	0.0
FreeMind [Bas09]	Java	90.0	**46.6**	11.1	0.0	2.2	0.0	28.8	11.1	0.0
MinGW-Minimalist GNU for Windows [Cot13]	C	60.0	**46.6**	10.0	0.0	38.3	0.0	0.0	5.0	0.0
Mirantis [This chapter]	Puppet		6.4	**49.3**	6.6	6.7	1.9	7.5	6.4	12.5
Mozilla [This chapter]	Puppet	558.0	7.7	**36.5**	6.4	17.0	2.3	10.0	1.9	8.4
Openstack [This chapter]	Puppet	1987.0	5.9	**57.5**	8.6	6.7	2.6	2.4	2.9	6.5
Wikimedia [This chapter]	Puppet	298.0	4.0	**54.5**	5.7	14.7	5.3	5.3	3.2	6.1

CHAPTER

5

OPERATIONS THAT CHARACTERIZE DEFECTIVE INFRASTRUCTURE AS CODE SCRIPTS

Prioritization of inspection and testing efforts is vital to maintain quality of IaC scripts. By identifying what operations are conducted in defective IaC scripts, we can provide recommendations for practitioners to efficiently allocate their inspection and testing efforts. In this chapter we identify operations that characterize defective IaC scripts. First, we provide related work, then we provide the methodology and the results of the study.

5.1 Related Work in Non-IaC Domain

Prior research has used text mining techniques such as BOW and TF-IDF to characterize and predict security and non-security defects. Scandariato et al. [Sca14] mined token frequencies from 20 Android applications and used the mined text features to predict vulnerabilities that appear in these applications. Walden et al. [Wal14] applied BOW-based text mining technique on web application source code to predict if the web applications contain vulnerabilities. They observed that for their selection of web applications text mining works better for vulnerability prediction than that of static code metrics. Perl et al. [Per15] applied the BOW model on commit messages extracted from version control repositories to predict security defects in 66 Github projects. Hovsepyan et al. [Hov12] mined text features from source code of 18 versions of a mobile application, and used the text features to predict vulnerable files. They reported an average precision and recall of 0.85 and 0.88. Mizuno et al. [Miz07] extracted patterns of tokens from source code using spam filter technique to predict fault prone modules for two projects: argUML and eclipse BIRT. Hata et al. [Hat10] mined text features on source code from 10 releases of five projects using the spam filter technique. They observed with logistic regression, text-based features outperform source code metrics with respect to building fault prediction models.

The above-mentioned studies demonstrate the use of text features in software defect analysis, and motivate us to extract text features via text mining techniques. The syntax and semantics of DSLs are fundamentally different from GPLs [Voe13] [Hud98] [VW07], and through systematic investigation we can determine if text-based features can be used effectively for characterizing and predicting defective IaC scripts.

We answer the following research questions:

- RQ1: What are the characteristics of defective infrastructure as code scripts? How frequently do the identified characteristics appear?
- RQ2: How can we build prediction models for defective infrastructure as code scripts using text features?

5.2 Methodology

We conduct our methodology by using the datasets described in Section 3. We state the research questions and the methodology in this section.

5.2.1 Text Preprocessing

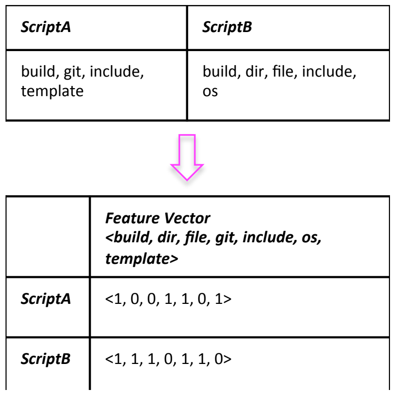
We apply text pre-processing in the following steps:

- First, we remove comments from scripts.
- Second, we split the extracted tokens according to naming conventions: camel case, pascal case, and underscore. These split tokens might include numeric literals and symbols, so we remove these numeric literals and symbols. We also remove stop words.
- Finally, we apply Porter stemming [Por97] on the collected tokens. After completing the text pre-processing step, we collect a set of pre-processed tokens for each IaC script in each dataset. We use these sets of tokens to create feature vectors, as will be discussed in Section 5.2.2.

5.2.2 Text Feature Extraction

We use two text mining techniques to extract text features: 'bag-of-words (BOW)' [Har54] and 'term frequency-inverse document frequency (TF-IDF)' [Man08]. The BOW technique, which has been extensively used in software engineering [Per15] [Wal14] [Hov12], converts each IaC script in the dataset to a set of words or tokens, along with their frequencies. Using the frequencies of the collected tokens, we create features. Similar to BOW, the TF-IDF technique is also popular in software engineering [Flo17] [EA17]. The TF-IDF technique accounts for the relative frequency of tokens that appear in scripts. Along with BOW, we also use TF-IDF, as prior research has demonstrated that TF-IDF can help in building better prediction models [Man08].

Bag-of-Words: Using the BOW technique, we use the tokens extracted from the text preprocessing steps of Section 5.2.1. We compute the occurrences of tokens for each script to construct a feature vector. Finally, for all the scripts in the dataset, we construct a feature matrix.



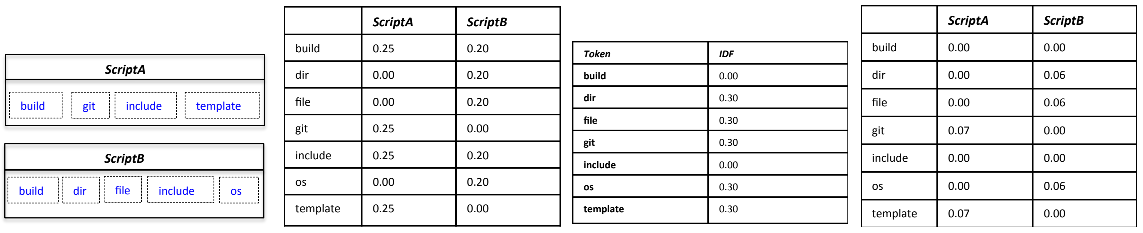
ScriptA	***ScriptB***
build, git, include, template	build, dir, file, include, os

	Feature Vector <build, dir, file, git, include, os, template>
ScriptA	<1, 0, 0, 1, 1, 0, 1>
ScriptB	<1, 1, 1, 0, 1, 1, 0>

Figure 5.1 A hypothetical example to illustrate the BOW technique discussed in Section 5.2.2.

We use a hypothetical example shown in Figure 5.1 to illustrate the BOW technique. In our example, our dataset has two IaC scripts $ScriptA$ and $ScriptB$ that respectively contain four and five pre-processed tokens. From the occurrences of tokens, we construct a feature matrix where the frequency of each token is represented as a vector. For example, according to Figure 5.1 the token 'build' appears once for $ScriptA$ and $ScriptB$.

TF-IDF: The TF-IDF technique computes the relative frequency of a token compared to other tokens, across all documents [Man08]. In our experimental setting, the tokens that we apply TF-IDF on are extracted from IaC scripts, as discussed in Section 5.2.1. For a script s, and a token t, we calculate the TF-IDF as following:


ScriptA
build
git
include
template
ScriptB
build
dir
file
include
os

(a)

	ScriptA	ScriptB
build	0.25	0.20
dir	0.00	0.20
file	0.00	0.20
git	0.25	0.00
include	0.25	0.20
os	0.00	0.20
template	0.25	0.00

(b)

Token	IDF
build	0.00
dir	0.30
file	0.30
git	0.30
include	0.00
os	0.30
template	0.30

(c)

	ScriptA	ScriptB
build	0.00	0.00
dir	0.00	0.06
file	0.00	0.06
git	0.07	0.00
include	0.00	0.00
os	0.00	0.06
template	0.07	0.00

(d)

Figure 5.2 A hypothetical example to illustrate out feature vectorization technique using TF-IDF. Figure 5.2a presents the pre-processed tokens of two scripts: $ScriptA$ and $ScriptB$. Figure 5.2b presents the TF values for both scripts. Figure 5.2c presents the IDF scores for the unique seven tokens. Finally, TF-IDF scores for each script and token is presented in Figure 5.2d.

- **Calculate TF**: We calculate TF of token t in script s using Equation 5.1:

$$TF(t,s) = \frac{\text{occurences of token } t \text{ in script } s}{\text{total count of tokens in script } s} \tag{5.1}$$

- **Calculate IDF**: We calculate IDF of token t using Equation 5.2:

$$IDF(t) = \log_{10}\left(\frac{\text{total count of scripts in the dataset}}{\text{count of scripts in which token } t \text{ appears at least once}}\right) \tag{5.2}$$

- **Calculate TF-IDF**: We calculate TF-IDF of token t using Equation 5.3:

$$TF-IDF(t,s) = TF(t,s) * IDF(t) \tag{5.3}$$

We use a hypothetical example to illustrate how the TF-IDF vectorization process works, as shown in Figure 5.2. In our example, our dataset has two IaC scripts $ScriptA$ and $ScriptB$ that respectively contain four and five pre-processed tokens. The total unique tokens in our hypothetical dataset is seven because two tokens appear in both scripts. Using Equation 5.1, we calculate the TF metric for each of these tokens and for both scripts: $ScriptA$ and $ScriptB$. For example, the TF metric for token 'template' and $ScriptA$ is 0.25, as the token 'template' appears once in $ScriptA$, and the total count of tokens in $ScriptA$ is 4. Next, we show the calculation of metric IDF for all tokens using Equation 5.2. For the token 'template' we observe IDF to be 0.3, as it appears in one of the two scripts in our hypothetical dataset. Finally, using Equation 5.3, we determine the TF-IDF scores for token 'template'. For $ScriptA$ and $ScriptB$ token 'template' has a TF-IDF score of 0.07, and 0.0, respectively.

Upon completion of this step we create a feature vector for each script in the dataset.

5.2.3 Feature Selection

To account for collinearity, we apply PCA as described in Section 2.2.

We use the identified components using PCA analysis to answer both RQs. As will be described in Section 5.2.4, to answer RQ1, we apply qualitative analysis on the text features that correlate with defective IaC scripts. Next, as will be described in Section 5.2.5 to answer RQ2, we use the identified principal components as input to statistical learners for building defect prediction models.

5.2.4 RQ1: What are the characteristics of defective infrastructure as code scripts? How frequently do the identified characteristics appear?

The text features included in the identified principal components from Section 5.2.3 are correlated with defective IaC scripts. However, these text features are tokens, which might be insufficient to produce actionable information for practitioners. We address this issue by applying qualitative analysis on the identified tokens. We apply a qualitative analysis called Strauss-Corbin Grounded Theory (SGT) [SC94]. SGT is a variant of Grounded Theory (GT) [SC94] [Sto16] that allows for specification of research questions and is used to characterize properties from textual artifacts [SC94] [Sto16] [AF17]. SGT includes three elements: ‘codes’, ‘low-level concept’, and ‘high-level conclusion’. In SGT, a ‘high-level conclusion’ represents an attribute or property [SC94], and by deriving these high-level conclusions, we identify properties that characterize defective scripts.

We use an example in Figure 5.3 to explain how we use the three SGT elements. We first start with text features that characterize defective IaC scripts determined by our PCA analysis to derive necessary codes. These codes are formed using text features that share a common attribute. In the example, we separate the text features into two codes: one code is related to directories, and the other code is related to files. Next, we generate low-level concepts from the codes by creating a higher level of abstraction than text features. For example, the low-level concept ‘directory-related action and attributes’ was determined by the 11 tokens identified as codes. The final step is to draw high-level conclusions by identifying similarities between the low-level concepts. In the example, the two low-level concepts are related to performing filesystem operations. We use these two low-level concepts to determine the observation ‘Filesystem operations appear more in defective IaC scripts’. The identified observation represents an operation that characterize defective IaC scripts which is, in defective IaC scripts more filesystem operations are performed.

The second part of RQ1 is focused on the frequency of the identified properties that characterize defective scripts. By quantifying the frequency of the identified properties, we can identify how many scripts can be prioritized for verification and validation (V&V) using that particular property. We determine the frequency of each property by counting for how many scripts the property appears at least once, in the following two-step process:

- **Step 1-Keyword Search**: First, we identify if any of the text features used as codes for a property, appears at least once in any of the IaC scripts. As a hypothetical example, if any of the following text features ‘dir’, and ‘file’, that are used as codes for a property, appear at least once in a script,

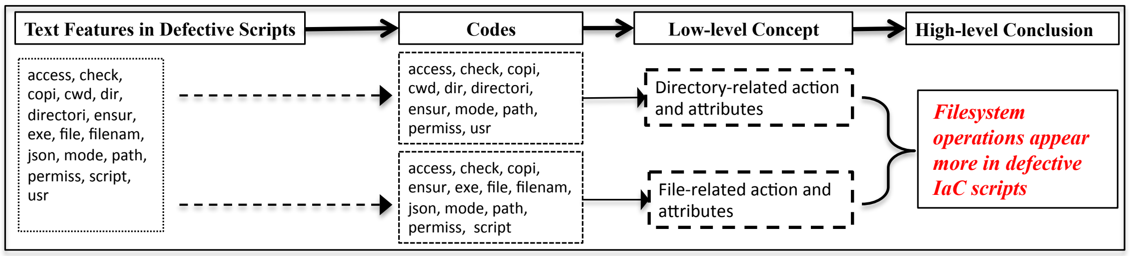

Text Features in Defective Scripts
Codes
Low-level Concept
High-level Conclusion
access, check, copi, cwd, dir, directori, ensur, exe, file, filenam, json, mode, path, permiss, script, usr
access, check, copi, cwd, dir, directori, ensur, mode, path, permiss, usr
access, check, copi, ensur, exe, file, filenam, json, mode, path, permiss, script
Directory-related action and attributes
File-related action and attributes
Filesystem operations appear more in defective IaC scripts

Figure 5.3 An example of how we use Strauss-Corbin Grounded Theory (SGT) to characterize defective IaC scripts.

then that script is considered for further analysis. Completion of Step 1 will provide a list of scripts which need further inspection in Step 2.

- **Step 2-Manual Examination**: The identified scripts in Step 1 can yield false positives, for example, a script can contain the text feature 'file', even though the script is unrelated with filesystem operations. We apply manual analysis to determine which scripts actually contain the property of interest. We consider a script to contain a property if:
  - the script uses the required IaC syntax to implement the property. (for example, to implement a filesystem operation in Puppet a script must use the 'file' syntax[1]); or
  - the comments in the script reveals the property of interest ('This script changes permission of file a.txt' is an example comment that reveals that the script performs a filesystem operation)

Upon completion of the above-mentioned two-step process, we will identify which properties appear in how many scripts.

5.2.5 RQ2: How can we build prediction models for defective infrastructure as code scripts using text features?

We answer RQ2 by using the principal components identified from Section 5.2.3. Next, we use statistical learners to construct defect prediction models. We evaluate the performance of the constructed defect prediction models using six measures mentioned in Section 2.1.

We apply two tuning strategies to investigate if tuning helps improve the performance of our constructed defect prediction models. The two tuning strategies are **tuning parameters of statistical learners** described in Section 2.1 and **tuning of text feature matrices**, described below.

[1]https://puppet.com/docs/puppet/5.3/types/file.html

5.2.5.0.1 Tuning of Text Feature Matrices

We take motivation from Krishna et al. [Kri16]'s findings where they reported the impact of text feature count on text feature-based prediction. They observed that upon construction of a TF-IDF matrix, improved prediction performance can be obtained by selecting the top-1000 text features with the highest TF-IDF scores. In this experiment, similar to Krishna et al. [Kri16], for each dataset we first sort the count-of-words and TF-IDF matrices. Then we select the top-i text features from the sorted matrices. We vary i from 250, 500, ... total number of text features existing in each dataset. Next, we construct defect prediction models using the selected text features and measure prediction performance with respect to the six metrics. We repeat the same process for both approaches: BOW and TF-IDF.

5.2.5.1 10×10-fold Cross Validation

We use 10×10-fold cross validation to evaluate the constructed prediction models. In the 10-Fold cross validation evaluation approach, the dataset is partitioned into 10 equal sized subsamples or folds [Tan05]. The performance of the constructed prediction models are tested by using nine of the 10 folds as training data, and the remaining fold as test data. Similar to prior work [Gho15], we repeat the 10-fold cross validation 10 times to assess prediction stability.

5.3 Results

In this section, we present answers to the two research questions.

5.3.1 RQ1: What are the characteristics of defective infrastructure as code scripts? How frequently do the identified characteristics appear?

We identify 2280, 3542, and 2398 unique text features from 2,138 IaC scripts for Mozilla, Openstack, and Wikimedia, respectively. By applying PCA analysis, we observe that respectively for Mozilla, Openstack, and Wikimedia 393, 437, and 327 components account for at least 95% of the total variance when BOW is applied. When TF-IDF is applied, we observe 557, 485, and 662 components account for 95% of the total variance. For both BOW and TF-IDF, the components identify text features that are correlated with defective scripts. As described in Section 5.2.4, we use these text features to derive properties that characterize defective IaC scripts using SGT. We identify three properties that characterize defective IaC scripts: 'filesystem operations', 'infrastructure provisioning', and 'managing user accounts'. These three properties are derived from text features using BOW and TF-IDF. Each of these properties corresponds to an operation executed in an IaC script. We list the identified properties that characterize defective IaC scripts with example code snippets in

Table 5.1. We list each property in the 'Characteristic' column, and a corresponding example code snippet in the 'Example Code Snippet' column. We briefly describe each property as following:

- ***Filesystem operations***: Filesystem operations are related to performing file input and output tasks, such as setting permissions of files and directories. For example, in Table 5.1 we report a code snippet that assigns permission mode '0444' to the file '/etc/firejail/thumbor.profile'. The file is assigned to owner 'root', and belongs to the group 'root'.

- ***Infrastructure provisioning***: This property relates to setting up and managing infrastructure for specialty systems, such as data analytics and database systems. From our qualitative analysis, we identify four types of systems that are provisioned: build systems, data analytics systems, database systems, and web server systems. Cito et al. [Cit15b] observed that IaC tools have become essential in cloud-based provisioning, and our finding provides further evidence to this observation. Vendors for IaC tools such as Puppet [2] advertise automated provisioning of infrastructure as one of the major capabilities of IaC tools, but our results indicate that the capability of provisioning via IaC tools can introduce defects.

- ***Managing user accounts***: This property of defective IaC scripts is associated with setting up accounts and user credentials. In Table 5.1, we provide an example on how user 'puppetsync' is created. One of the major tasks of system administrators is to setup and manage user accounts in systems [And02]. IaC tools, such as Puppet, provide API methods to create and manage users and their credentials in the system. According to some practitioners [Kel12], IaC tools, such as Puppet, can only be beneficial for managing a small number of users, and managing large number of users increases the chances of introducing defects in scripts.

As described in Section 5.2.4, we apply a two-step process to calculate the frequency of the properties that characterize defective scripts. After executing Step-1 (Keyword Search), we identify 37.5%, 51.0%, and 31.3% of Mozilla scripts to contain the property filesystem operations, infrastructure provisioning, and managing user accounts. For Openstack, we identify 50.9%, 42.1%, and 65.0% of the scripts to contain the three properties. For Wikimedia, we identify 67.2%, 26.7%, and 41.9% of the scripts respectively, to contain the properties: filesystem operations, infrastructure provisioning, and managing user accounts. After completion of Step 2 (Manual Examination), we report the frequency of identified properties that characterize defective IaC scripts in Table 5.2. The 'Characteristics' column represents a property, and in the 'Frequency' column we report the frequency of each dataset. We observe that for Mozilla 21.7% of scripts contain filesystem operations. The 'Infrastructure provisioning (total)' row presents the summation of the four provisioning-related operations: provisioning of (i) build, (ii) data analytics, (iii) database, and (iv) web server systems.

[2]https://puppet.com/products/capabilities/automated-provisioning

Table 5.1 Characteristics of Defective IaC Scripts

Characteristic	Example Code Snippet

Filesystem operations	`file {'/etc/firejail/thumbor.profile':`
	`  ensure => present,`
	`  owner  => 'root',`
	`  group  => 'root',`
	`  mode   => '0444',`
	`  source => 'puppet:///modules/thumbor/thumbor.profile',`
	`}`

Infrastructure provisioning	*Build systems*
	`exec {`
	`    'add-builder-to-mock_mozilla':`
	`    command => "/usr/bin/gpasswd -a ${users::builder::username} mock_mozilla",`
	`    unless => "/usr/bin/groups ${users::builder::username} | grep`
	`           '\\<mock_mozilla\\>'",`
	`    require => [Class['packages::mozilla::mock_mozilla'], Class['users::builder']];`
	`}`
	Data analytics systems
	`service {'elasticsearch':`
	`  ensure => running,`
	`  enable => true,`
	`  require => [`
	`     Package['elasticsearch', 'openjdk-7-jre-headless'],`
	`     File['/var/run/elasticsearch/'],`
	`  ]`
	`}`
	Database systems
	`mysql::user { $extension_cluster_db_user:`
	`   password => $extension_cluster_db_pass,`
	`   grant    => "ALL PRIVILEGES ON ${extension\_cluster\_shared\_db\_name}.*"`
	`}`
	Web server systems
	`file{'/etc/apache2/ports.conf':`
	`  content => template('apache/ports.conf.erb'),`
	`  require => Package['apache2'],`
	`  notify => Service['apache2'],`
	`}`

Managing User Accounts	`user {`
	`    'puppetsync':`
	`    managehome => true,`
	`    home => $homedir,`
	`    password => '*', # unlock the account without setting a password`
	`    comment => "synchronizes data between puppet masters";`
	`}`

Table 5.2 Frequency of Identified Operations Based on Keyword Search and Manual Examination

Characteristics	**Frequency**			
	MIR	MOZ	OST	WIK
Filesystem operations	15.1%	21.7%	14.5%	23.4%
Infrastructure provisioning (build systems)	0.5%	2.8%	0.0%	0.0%
Infrastructure provisioning (data analytics systems)	1.1%	2.7%	6.2%	5.4%
Infrastructure provisioning (database systems)	10.5%	0.8%	7.7%	4.3%
Infrastructure provisioning (web server systems)	17.0%	0.6%	5.0%	8.2%
Infrastructure provisioning (total)	29.1%	6.9%	18.9%	17.9%
Managing user accounts	6.6%	2.5%	1.1%	1.6%
Total	50.8%	31.1%	34.5%	42.9%

Table 5.2 also indicates how many scripts can be prioritized for V&V efforts. For example, considering filesystem operations for Mozilla, 21.7% of the total scripts can be prioritized. As shown in the 'Total' row, then instead of using all 299 IaC scripts for V&V, 180 (31.1%) of them can be prioritized. Similarly, considering all three properties, 34.5% and 42.9% of all scripts respectively in Openstack and Wikimedia can be prioritized for V&V.

5.3.2 RQ2: How can we build prediction models for defective infrastructure as code scripts using text features?

Using the steps described in Section 2.2.3, we obtain the list of parameters that need to be tuned. By executing **Step-1**, we collect 10, 2, 44, and 12 publications from IEEEXplore, ACM Digital Library, ScienceDirect, and Springer, respectively. From this total collection of 68 publications, we observe nine publications to report their use of statistical learner with at least one parameter (**Step-2**). Finally, in **Step-3**, we observe three of the nine publications, to use CART, LR, or RF. From these publications, we determine what parameters need to be tuned and the range of values for each parameter. Of the reported parameters, we only select those parameters for tuning which are available as part of the Scikit Learn API [Ped11].

In Table 5.3, we report the parameters we selected for tuning using DE. The 'Parameter' column presents the parameter name followed by the reference of the publication from which we derive the parameter and the value range. The 'Description' and 'Range' columns, respectively, describe a short description of the parameter and the range of values for the parameter. All parameters are numeric except for 'penalty', a parameter of LR. 'penalty' is string-based.

The median AUC values are presented in Table 6.9. The 'BOW' and 'TF-IDF' columns provides the median AUC values, respectively, for the BOW and TF-IDF technique. For AUC the BOW technique outperforms the TF-IDF technique for all four datasets.

We report the median AUC, precision, recall, F-measure, accuracy, and g-mean values for 10×10

Table 5.3 Parameters Selected for Tuning

Learner	Parameter	Description	Range
CART	*max_depth* [Fu16]	Maximum depth of the tree. By default, CART expands the tree until all leaves belong to only one class [Ped11].	[1, 50]
	max_features [Fu16]	Number of features to consider for the best split. By default, CART uses all features to build a tree [Ped11].	[0.01, 1.0]
	min_samples_split [Fu16]	Minimum number of samples required to split an internal node. Default: 2	[2, 20]
	min_samples_leaf [Fu16]	Minimum number of samples required to be at a leaf node. Default: 1	[1, 20]
LR	λ [EK13]	Penalty to increase the magnitude of features to reduce overfitting. Default: 1.0	[0.01, 1.0]
	penalty [EK13]	Penalty function. Default: ‘l2’	‘l1’,‘l2’
NB	Distribution [MN98] [Zha04]	Assumption on the underlying probabilistic distribution. Default: Gaussian	‘Gaussian()’, ‘Bernoulli()’
RF	*max_features* [Fu16]	Number of features to consider for the best split. By default, RF uses square root of features to build a tree [Ped11].	[0.01, 1.0]
	max_leaf_nodes [Fu16]	Grow trees with *max_leaf_nodes* in best first fashion. By default, RF does not limit the count of nodes to grow trees [Ped11].	[1, 50]
	min_samples_split [Fu16]	Minimum number of samples required to split an internal node. Default: 2	[2, 20]
	min_samples_leaf [Fu16]	Minimum number of samples required to be at a leaf node. Default: 1	[1, 20]
	n_estimators [Fu16]	Number of trees in the forest. Default: 100	[50, 150]

Table 5.4 Median AUC for two model building techniques: BOW and TF-IDF.

Dataset	**BOW**				**TF-IDF**			
	CART	**LR**	**NB**	**RF**	**CART**	**LR**	**NB**	**RF**
MIR	0.69	0.61	0.64	**0.73**	0.66	0.66	0.61	0.71
MOZ	0.61	0.64	0.52	**0.65**	0.58	0.58	0.51	0.53
OST	0.58	**0.63**	0.61	0.59	0.54	0.57	0.57	0.55
WIK	0.63	0.62	**0.68**	0.67	0.56	0.56	0.55	0.56

cross validation, for all learners and all datasets respectively in Tables 6.9, 6.10, 6.11, 6.12, 5.8, and 5.9. With respect to median accuracy, precision, and F-measure, the BOW technique outperforms the TF-IDF technique for three datasets. The TF-IDF technique outperforms the BOW technique for three datasets with respect to median recall.

5.3.2.1 Building Prediction Models By Tuning Text Feature Matrix

We present our findings on how tuning text feature matrix can impact prediction performance for the two techniques: BOW and TF-IDF.

Table 5.5 Median precision for two model building techniques: BOW and TF-IDF.

Dataset	**BOW**				**TF-IDF**			
	CART	**LR**	**NB**	**RF**	**CART**	**LR**	**NB**	**RF**
MIR	0.69	0.63	**0.75**	0.72	0.68	0.66	0.64	0.72
MOZ	0.61	0.64	0.63	0.65	0.56	0.51	0.46	**0.67**
OST	0.68	0.71	**0.76**	0.66	0.62	0.64	0.64	0.62
WIK	0.66	0.75	**0.76**	0.68	0.59	0.58	0.59	0.58

Table 5.6 Median recall for two model building techniques: BOW and TF-IDF.

Dataset	**BOW**				**TF-IDF**			
	CART	**LR**	**NB**	**RF**	**CART**	**LR**	**NB**	**RF**
MIR	0.76	0.53	0.67	0.84	0.74	**0.85**	0.68	0.84
MOZ	**0.61**	0.41	0.51	0.65	0.56	0.39	0.53	0.43
OST	0.67	0.61	0.71	0.70	0.72	**1.00**	0.71	0.98
WIK	0.68	0.47	0.75	0.80	0.69	**1.00**	0.94	0.82

Table 5.7 Median F-measure for two model building techniques: BOW and TF-IDF.

Dataset	**BOW**				**TF-IDF**			
	CART	**LR**	**NB**	**RF**	**CART**	**LR**	**NB**	**RF**
MIR	0.71	0.59	0.67	**0.77**	0.67	0.72	0.65	0.72
MOZ	0.57	0.49	0.49	**0.64**	0.54	0.44	0.49	0.49
OST	0.65	0.65	0.69	0.67	0.69	0.74	0.66	**0.74**
WIK	0.66	0.56	0.71	**0.72**	0.59	0.70	0.71	0.65

Table 5.8 Median accuracy for two model building techniques: BOW and TF-IDF.

Dataset	**BOW**				**TF-IDF**			
	CART	**LR**	**NB**	**RF**	**CART**	**LR**	**NB**	**RF**
MIR	0.67	0.60	0.65	0.67	0.65	0.67	0.61	**0.71**
MOZ	0.62	0.62	0.63	**0.67**	0.58	0.54	0.51	0.61
OST	0.59	**0.63**	0.63	0.60	0.56	0.60	0.58	0.60
WIK	0.62	0.60	0.67	**0.68**	0.56	0.57	0.62	0.55

Table 5.9 Median G-mean for two model building techniques: BOW and TF-IDF.

Dataset	**BOW**				**TF-IDF**			
	CART	**LR**	**NB**	**RF**	**CART**	**LR**	**NB**	**RF**
MIR	0.67	0.70	0.67	0.69	0.67	**0.73**	0.65	0.76
MOZ	0.59	0.50	0.49	**0.64**	0.55	0.45	0.49	0.51
OST	0.65	0.66	0.69	**0.76**	0.69	**0.76**	0.66	0.75
WIK	0.71	0.59	0.71	**0.76**	0.60	0.73	0.73	0.66

5.3.2.1.1 Tuning Text Feature Matrix with the Bag-of-words Technique

When we apply the text feature matrix tuning technique, we observe the impact on prediction performance. We present our findings in Figures 5.4 to 5.7. In Figures 5.4, 5.5, 5.6, and 5.7 we, respectively, present the impact of tuning text feature matrix on prediction performance for Mirantis, Mozilla, Openstack, and Wikimedia. In each of these figures, we report the prediction performance for the four learners CART, LR, NB, and RF. In each figure, the x-axis corresponds to the varying amount of text features used to construct the text feature matrix, and y-axis presents the median prediction performance values. We use these figures to demonstrate how tuning of text features impacts prediction performance i.e. show increasing, decreasing, or constant trends.

The impact of tuning text feature matrix however is different for different learners and the datasets. For the Mirantis dataset, for LR and NB the recall score decreases, but increases for CART. In case of the Mozilla dataset, when we use LR, the F-measure, G-mean, precision and recall values decrease when all the text features are used. In a similar manner, we observe when we use Wikimedia with LR, the values decrease for accuracy, AUC, F-measure, G-mean, precision, and recall.

For the Openstack dataset, we observe minor differences in prediction performance measures for all learners. However, this finding indicates that using smaller amount of text features can be used to obtain prediction performance measure similar to that of large amount of text features.

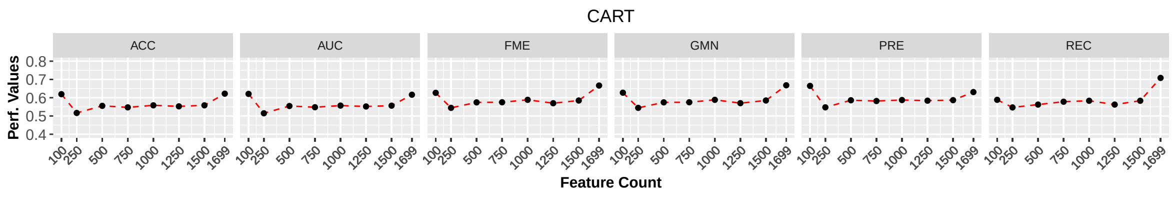
5.3.2.1.2 Tuning Text Feature Matrix with the TF-IDF Technique

With the TF-IDF technique, we also observe the impact of tuning text feature matrix on prediction performance. We present our findings in Figures 5.8- 5.11. In Figures 5.8, 5.9, 5.10, and 5.11 we, respectively, present the impact of tuning text feature matrix on prediction performance for Mirantis, Mozilla, Openstack, and Wikimedia. In each of these figures, we report the prediction performance for the four learners CART, LR, NB, and RF.

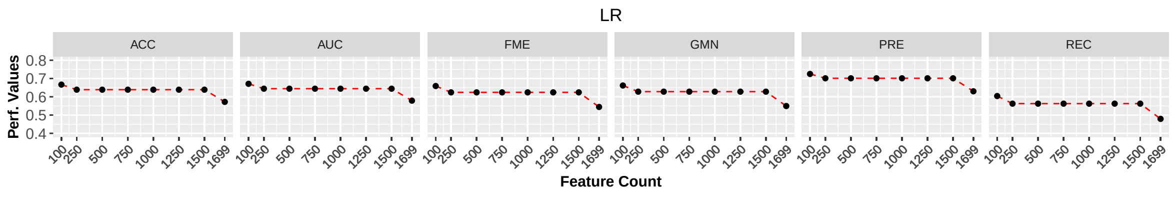
The impact of tuning text feature matrix is observable for specific learners. In the case of TF-IDF analysis, the change in precision and recall is observable for NB. For example, as shown in Figure 5.8, for NB the precision decreases from 0.78 to 0.64. In a similar manner, from Figure 5.9, we observe for LR and NB we observe precision to decrease. For LR, the precision when we use 1500 and 1808 text features is respectively 0.70 and 0.46. For NB, the precision when we use 1500 and 1808 text features is respectively 0.67 and 0.44.

For Mozilla, Openstack and Wikimedia, we notice NB to achieve the highest median recall when all text features in the dataset is used, indicating that even though false positives are generated (low precision), using all text features can be helpful to identify defective scripts in the dataset with the TF-IDF approach.

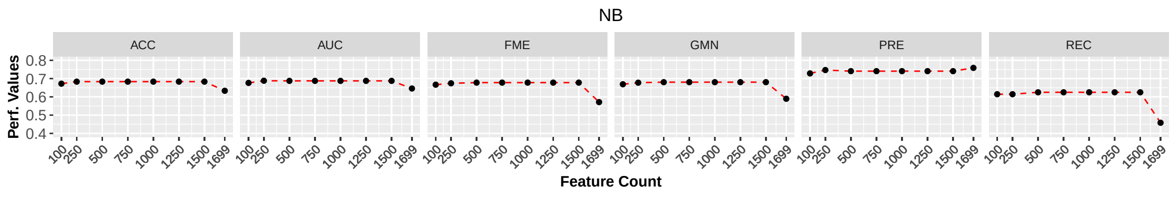
Another finding we observe from the Openstack dataset is that the prediction performance measures remain overall consistent. For example, when we use CART and vary the number of text



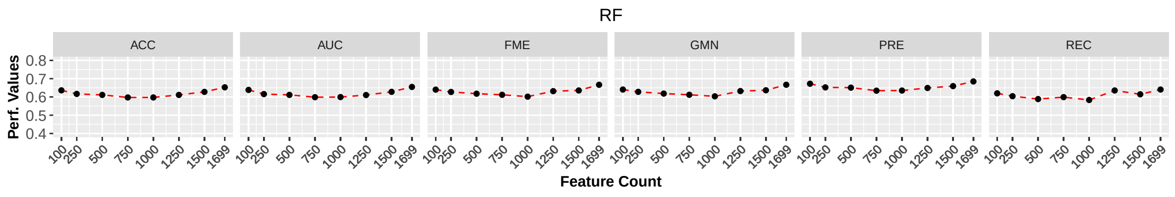
(a)



(b)

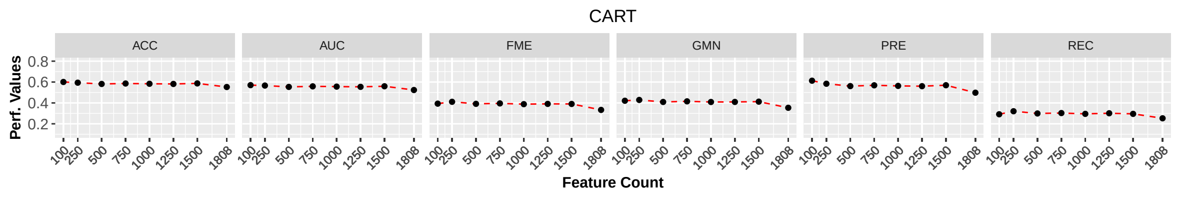


(c)

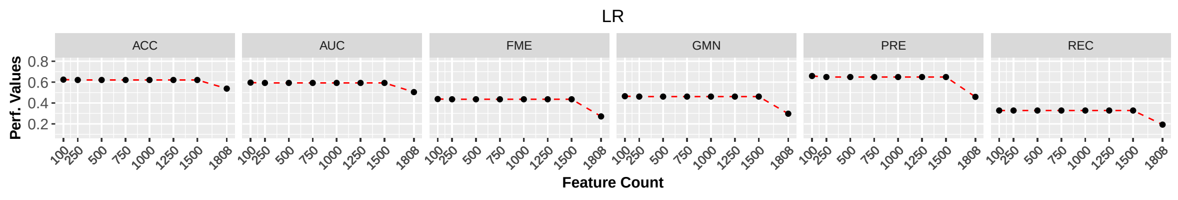


(d)

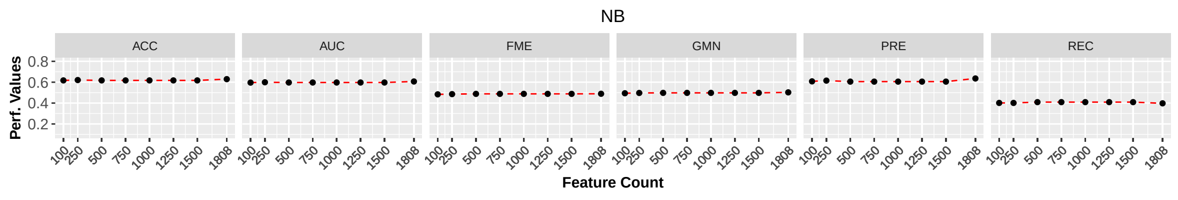
Figure 5.4 The impact of tuning text feature matrix on prediction performance for Mirantis when we use bag-of-words. Figures 5.4a, 5.4b, 5.4c, and 5.4d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.



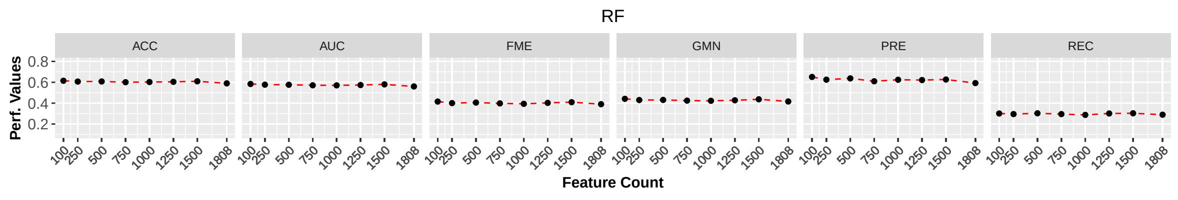
(a)



(b)

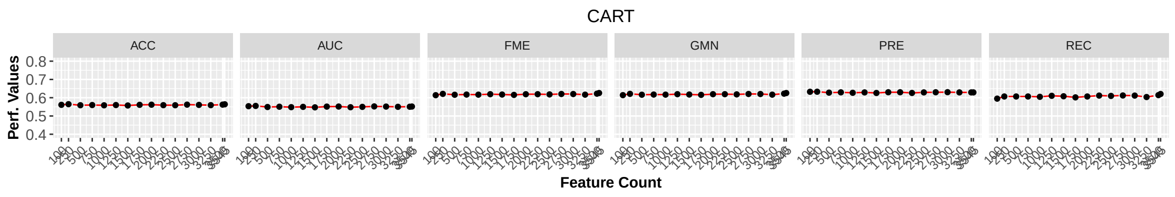


(c)

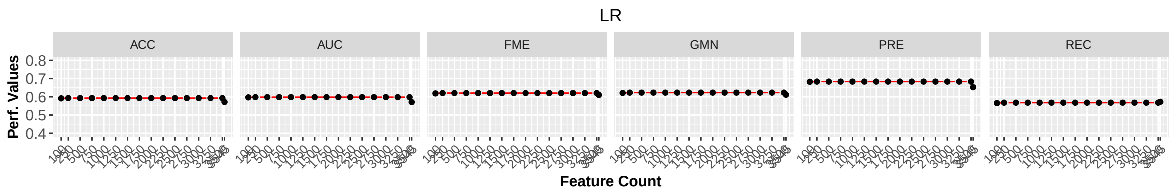


(d)

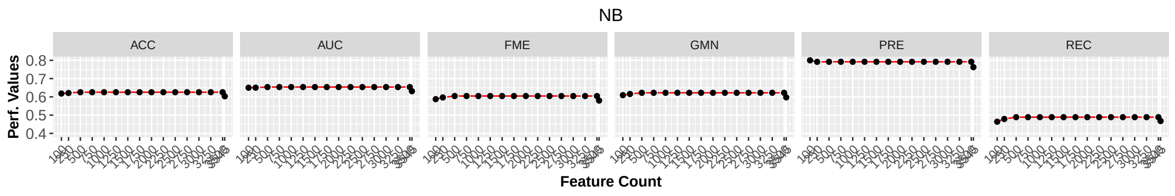
Figure 5.5 The impact of tuning text feature matrix on prediction performance for Mozilla when we use bag-of-words. Figures 5.5a, 5.5b, 5.5c, and 5.5d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.



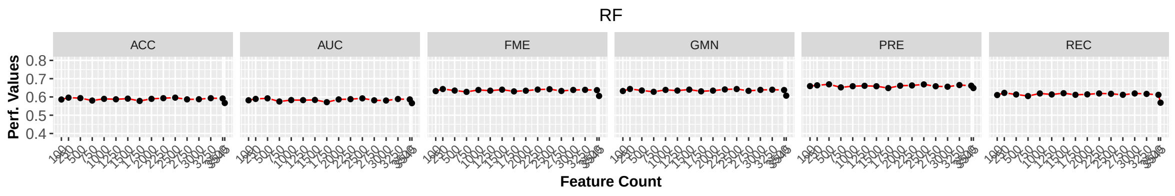
(a)



(b)

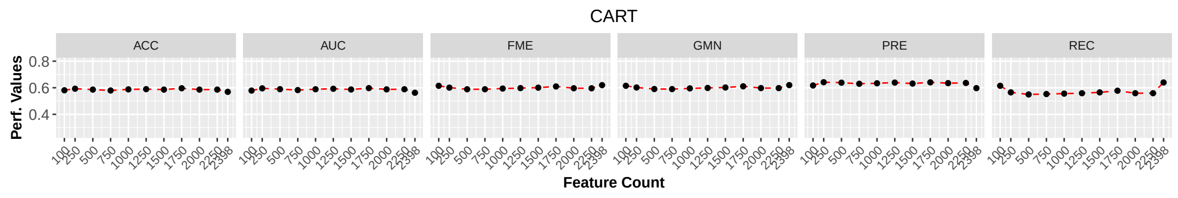


(c)

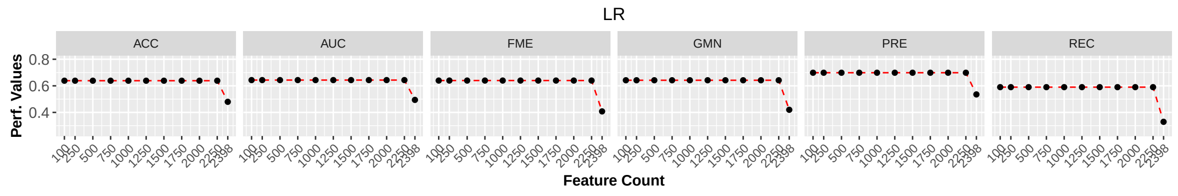


(d)

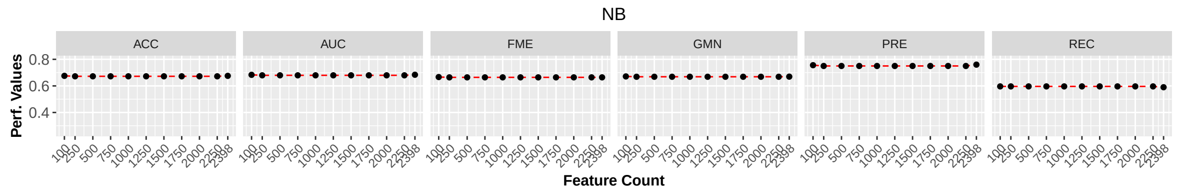
Figure 5.6 The impact of tuning text feature matrix on prediction performance for Openstack when we use bag-of-words. Figures 5.6a, 5.6b, 5.6c, and 5.6d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.



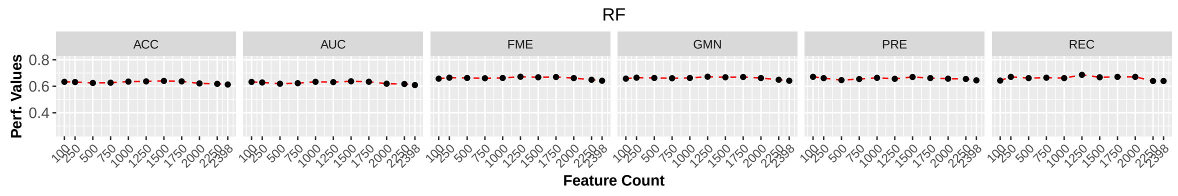
(a)



(b)



(c)



(d)

Figure 5.7 The impact of tuning text feature matrix on prediction performance for Wikimedia when we use bag-of-words. Figures 5.7a, 5.7b, 5.7c, and 5.7d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.

Table 5.10 The technique for which we observe the highest median prediction performance for each dataset.

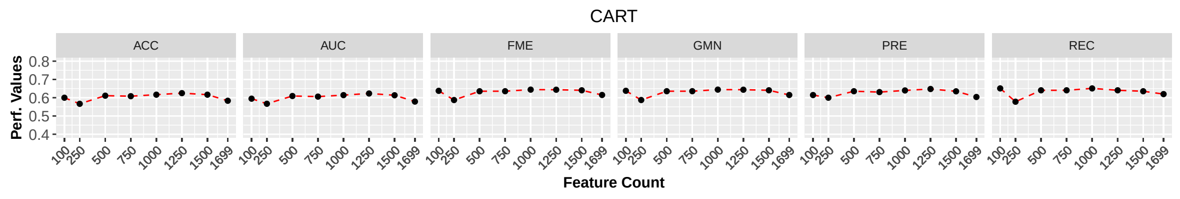
	MIR	**MOZ**	**OST**	**WIK**
Acc.	NB,TFIDF,TFM	RF,BOW,DE	NB/LR,BOW,DE	RF,BOW,DE
AUC	RF,BOW,DE	RF,BOW,DE	NB,BOW,TFM	NB,BOW,DE
F-Mea.	RF,BOW,DE	RF,BOW,DE	LR/RF,TFIDF,DE	RF,BOW,DE
G-Mean	RF,TFIDF,DE	RF,BOW,DE	RF,BOW,DE	RF,BOW,DE
Pre.	NB,TFIDF,TFM	LR,TFIDF,TFM	NB,TFIDF,TFM	NB,TFIDF,TFM
Recall	LR,TFIDF,DE	RF,BOW,DE	LR,TFIDF,DE	LR,TFIDF,DE

features selected from 100 to 3545, we observe AUC to vary between 0.51 and 0.55. The highest median AUC is observable when we use 100 text features. This finding indicates that using smaller amount of text features we may obtain the highest possible AUC for a certain learner.

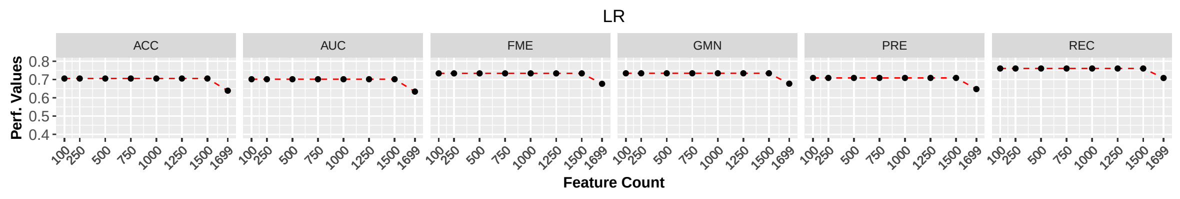
Summary: We summarize our findings from Sections 5.3.2 and 5.3.2.1 in Table 5.10. In Table 5.10, we report the combination of techniques for which we observe the highest median prediction performance. Each cell in the entry is a tuple where we report the learner, feature extraction technique, and the tuning technique i.e. DE or text feature matrix (TFM). For example, for the Mirantis dataset we observe the highest median accuracy when the learner, feature extraction technique, tuning technique is respectively, Naive Bayes (NB), TF-IDF, and text feature matrix (TFM).

From Table 5.10, we observe that for F-measure and Recall the DE tuning technique provides the best performance for all datasets. For precision we observe the highest performance with respect to precision for all four datasets. We observe BOW technique to co-occur more with the DE-based tuning technique compared to the text feature matrix technique. For precision, the TF-IDF technique co-occurs more with the text feature matrix technique instead of BOW.

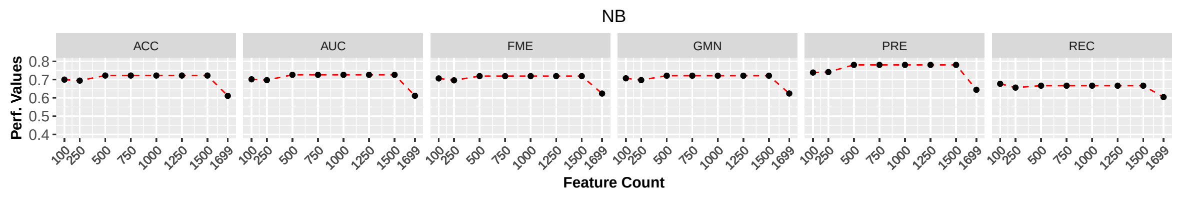
Summary: In short, by applying text mining techniques, and qualitative analysis we identify three properties that characterize defective scripts: filesystem operations, infrastructure provisioning, and managing user accounts. We also use the text mining techniques to construct defect prediction models. We observe based on prediction performance measure tuning techniques can impact differently. Along with parameter tuning of statistical learners we also observe the importance of tuning text feature matrices when text features are used. In our datasets, the distribution of defective scripts is around 50% but the prediction accuracy is around 70%, which suggests that the prediction performance of our constructed models could be improved.



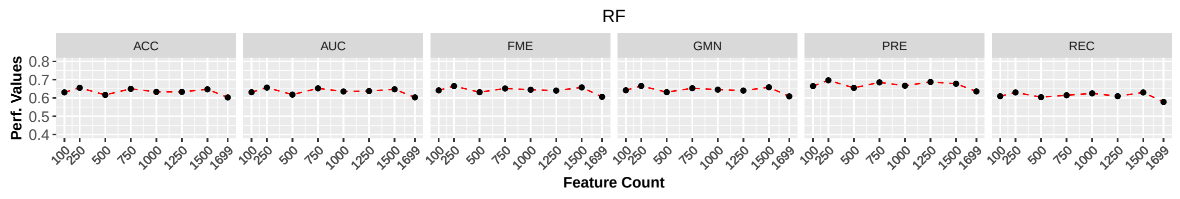
(a)



(b)

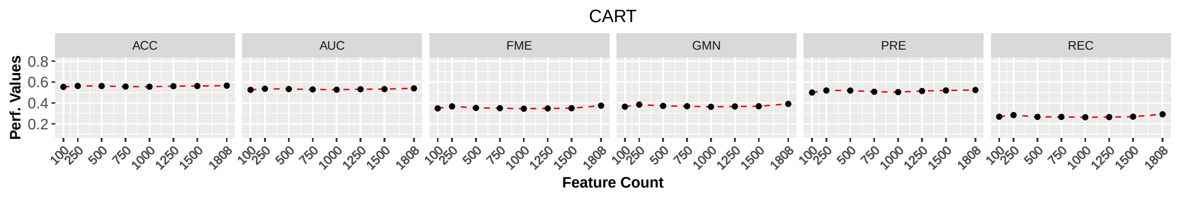


(c)

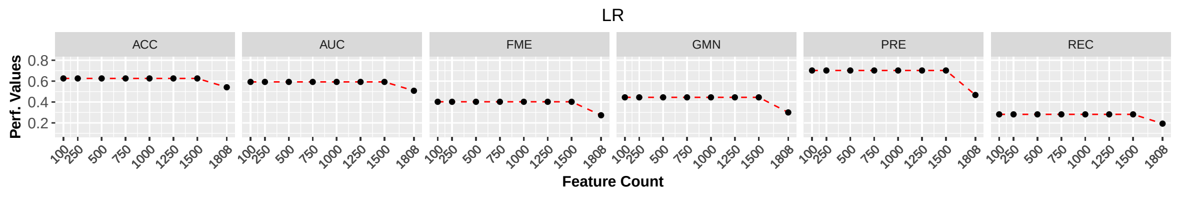


(d)

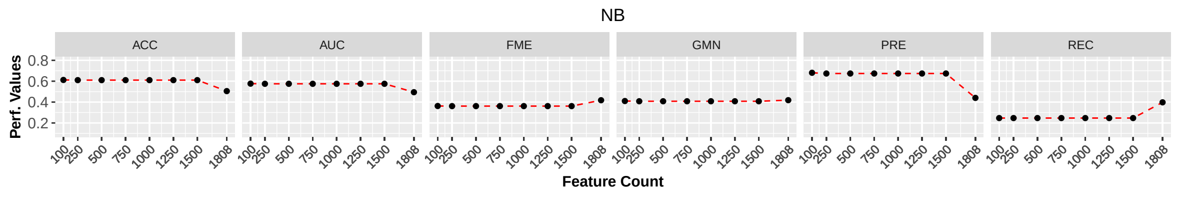
Figure 5.8 The impact of tuning text feature matrix on prediction performance for Mirantis when we use TF-IDF. Figures 5.8a, 5.8b, 5.8c, and 5.8d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.



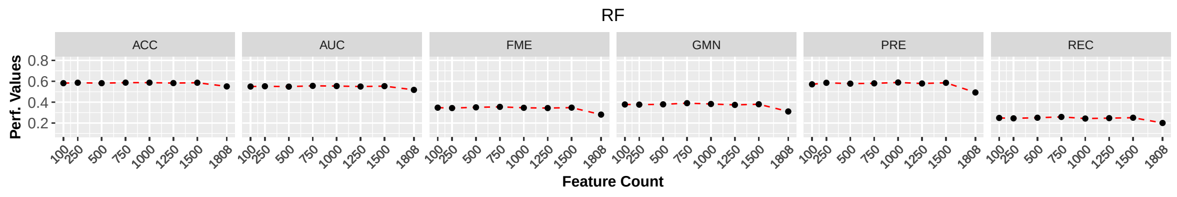
(a)



(b)

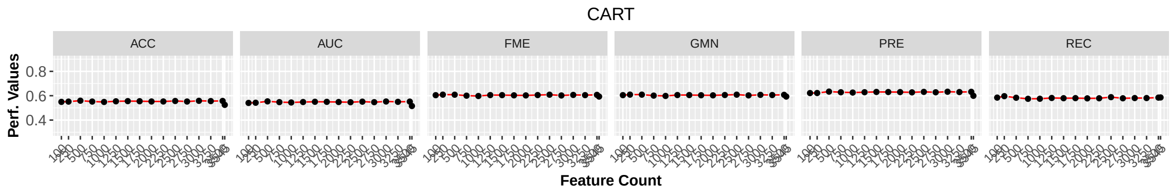


(c)

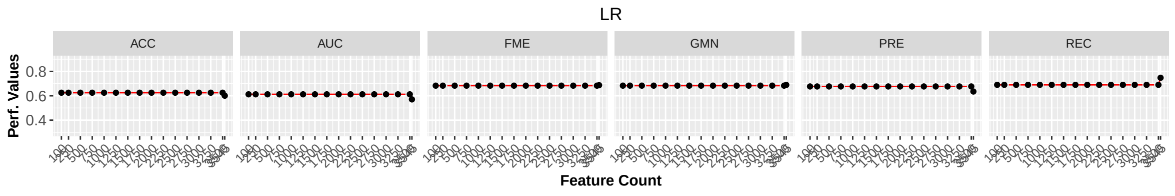


(d)

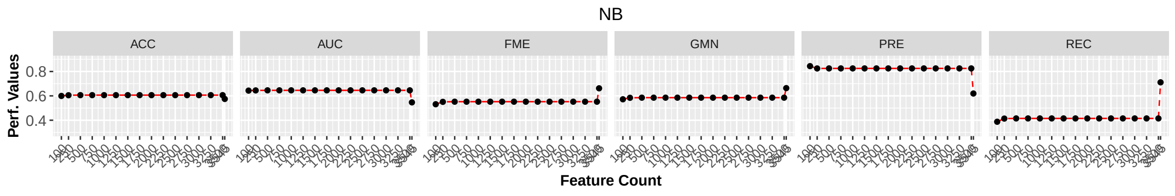
Figure 5.9 The impact of tuning text feature matrix on prediction performance for Mozilla when we use TF-IDF. Figures 5.9a, 5.9b, 5.9c, and 5.9d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.



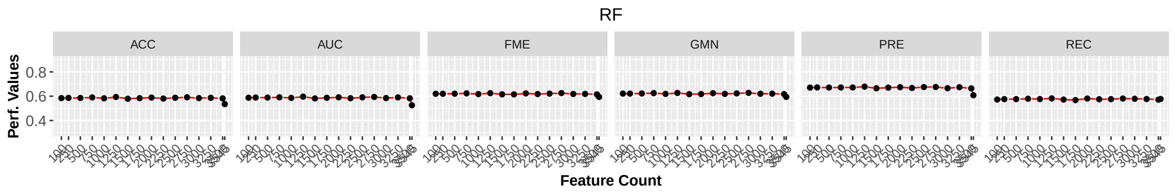
(a)



(b)

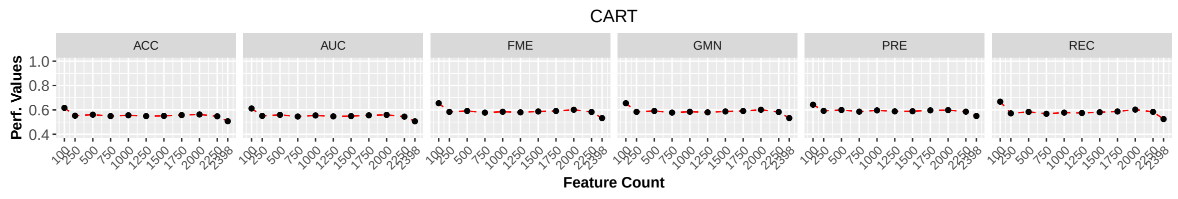


(c)

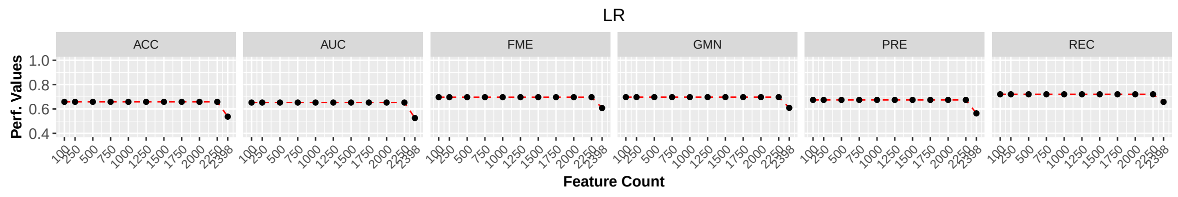


(d)

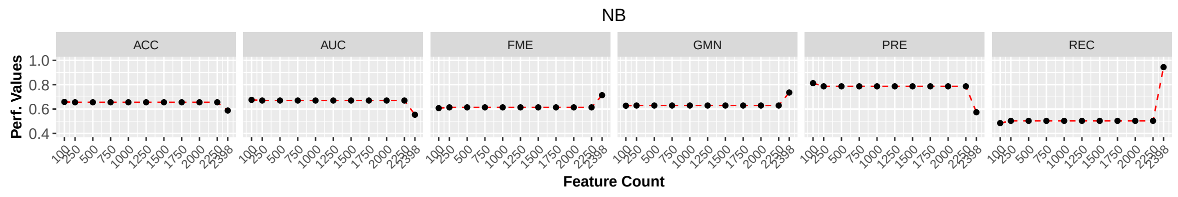
Figure 5.10 The impact of tuning text feature matrix on prediction performance for Openstack when we use TF-IDF. Figures 5.10a, 5.10b, 5.10c, and 5.10d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.



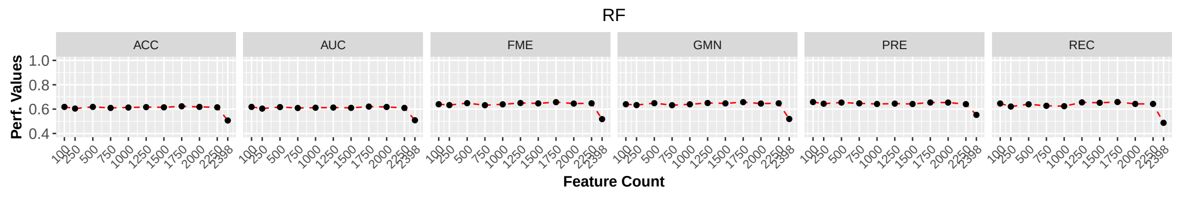
(a)



(b)



(c)



(d)

Figure 5.11 The impact of tuning text feature matrix on prediction performance for Wikimedia when we use TF-IDF. Figures 5.11a, 5.11b, 5.11c, and 5.11d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.

CHAPTER

6 SOURCE CODE PROPERTIES THAT CORRELATE WITH DEFECTIVE INFRASTRUCTURE AS CODE SCRIPTS

We hypothesize certain source code properties are correlated with defective IaC scripts. Through systematic investigation, we can identify a set of source code properties that correlate with defective scripts. Practitioners may benefit from our investigation as they can allocate sufficient inspection and testing efforts for the identified set of source code properties in IaC scripts. This chapter focuses on identifying source code properties that can be extracted from static analysis of IaC scripts, and correlated with IaC defects. We organize this chapter by first providing related work. Next, we provide methodology. Finally, we provide results of the study.

6.1 Related Work in Non-IaC Domain

Nagappan and Ball [NB05] investigated seven absolute code properties and eight relative code churn properties, and reported that relative code churn properties are better predictors of defect density. Zheng et al. [Zhe06b] investigated how static analysis can be used to identify defects in a large scale industrial software system. They [Zhe06b] observed that automated static analysis is a relatively affordable fault detection technique compared to that of manual inspection. Zimmermann et al. [Zim07] proposed a set of 14 static code properties for predicting defects in Eclipse and reported a precision and recall of 0.63~0.78, and 0.61~0.78, respectively. Ghotra et al. [Gho15] investigated which statistical learners can be used efficiently for defect prediction, and concluded that ensemble methods such as bagging, rotation forest, and random subspace can produce efficient defect prediction models. Rahman et al. [Rah14] investigated if defect prediction models are effective in finding defects compared to that of static bug finders such as, Findbugs and PMD. They observed that static bug finders frequently perform better when ordering their warnings using priorities produced by defect prediction models. Nunuez-Varela et al. [NV17] performed a systematic literature review of 226 research papers, and reported that researchers have extensively studied OOP metrics compared to that of aspect and feature oriented metrics for defect and fault prediction.

The above-mentioned research studies highlight the prevalence of source code properties that correlate with defects in source code. All of the above-mentioned research is applicable for GPLs, and may not be applicable for IaC scripts, which are DSLs and have their own syntax and semantics. We take motivation from these studies and investigate which source code properties correlate with defective IaC scripts.

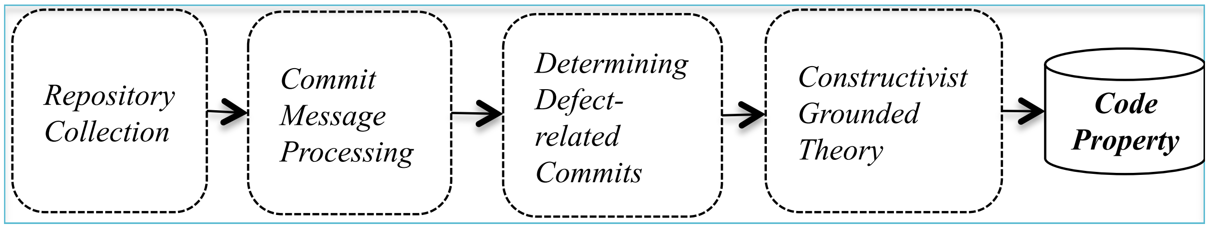
We analyze the following research questions:

- RQ1: What source code properties characterize defective infrastructure as code scripts?
- RQ2: Do practitioners agree with the identified source code properties?
- RQ3: How can we construct defect prediction models for infrastructure as code scripts using the identified source code properties?

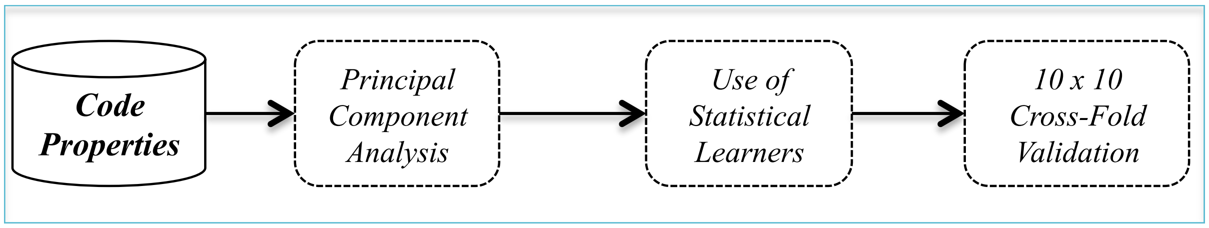
6.2 Methodology

6.2.1 RQ1: What source code properties characterize defective infrastructure as code scripts?

As the first step, we identify source code properties by applying qualitative analysis called constructivist grounded theory [Cha14]. We use defect-related XCMs and the code changes performed in defect-related commits that we determined in Section 3.1.4, to perform constructivist grounded theory. We use the defect-related XCMs because these messages can provide information on how to identify source code properties that are related to defects. Using only defect-related commit messages may not capture the full context to determine defect-related commits, so we also use code changes (commonly referred to as 'diffs' or 'hunks') from defect-related commits. We use defect-related commits because defect-related commits report what properties of the IaC source code are changed and whether or not the changes were adding or deleting code [Ala08].


Repository Collection
Commit Message Processing
Determining Defect-related Commits
Constructivist Grounded Theory
Code Property

(a)


Code Properties
Principal Component Analysis
Use of Statistical Learners
10 x 10 Cross-Fold Validation

(b)

Figure 6.1 Methodology: Figures 6.1a and 6.1b respectively summarizes the methodology for RQ1 and RQ3.

Grounded theory and its variants generally include three elements: ‘concepts’, ‘categories’, and ‘propositions’ [Pan96]. By deriving propositions, we identify properties and the description behind the identified properties. We use Figure 6.2 to explain how we use the three grounded theory elements to identify a property. We first start with defect-related XCMs and code changes from defect-related commits, to derive concepts. According to Figure 6.2, from the defect-related XCM ‘fix file location for interfaces change-id i0b3c40157’, we extract the concept ‘fix file location’. Next, we generate categories from the concepts, for example, we use the concept ‘fix file location’ to determine the category which states an erroneous file location might need fixing. We use three concepts to derive category ‘Path to external file or script needs fixing’. Finally, we use the categories ‘File location needs fixing’ and ‘Path to external file or script needs fixing’ to derive a proposition related to file location. This proposition gives us a property ‘File’ and the description behind that property is ‘Scripts that set file paths can be defect-prone’.

Upon completion of constructivist grounded theory, we obtain a set of source code properties. We extract the count of each identified property using Puppeteer [Sha16b]. We use the Mann-Whitney U test [MW47] to compare the property count for defective and neutral files. The null hypothesis is: the property is not different between defective and neutral files. The alternative hypothesis is: property is larger for defective files than neutral files. We consider a significance level of 95%. If $p-value < 0.05$, we reject the null hypothesis, and accept the alternative hypothesis.

Along with Mann-Whitney U test, we also apply Cliff’s Delta [Cli93] to compare the distribution of each characteristic between defective and neutral files. Both, Mann-Whitney U test and Cliff’s Delta

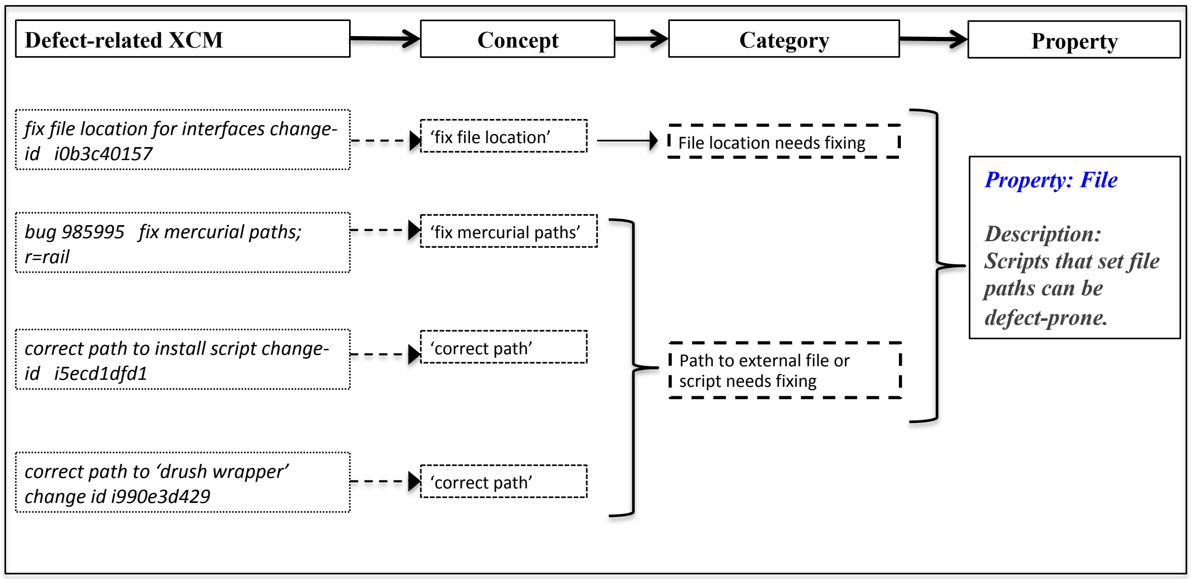

Defect-related XCM
Concept
Category
Property
fix file location for interfaces change-id i0b3c40157
bug 985995 fix mercurial paths; r=rail
correct path to install script change-id i5ecd1dfd1
correct path to 'drush wrapper' change id i990e3d429
'fix file location'
'fix mercurial paths'
'correct path'
'correct path'
File location needs fixing
Path to external file or script needs fixing
Property: File
Description: Scripts that set file paths can be defect-prone.

Figure 6.2 An example of how we identify source code properties using constructivist grounded theory.

are non-parametric. The Mann-Whitney U test states if one distribution is significantly large/smaller than the other, whereas effect size using Cliff's Delta measures how large the difference is.

We use Romano et al. [Rom06]'s recommendations to interpret the observed Cliff's Delta values. According to Romano et al. [Rom06], the difference between two groups is 'large' if Cliff's Delta is greater than 0.47. A Cliff's Delta value between 0.33 and 0.47 indicates a 'medium' difference. A Cliff's Delta value between 0.14 and 0.33 indicates a 'small' difference. Finally, a Cliff's Delta value less than 0.14 indicates a 'negligible' difference.

Relative Correlation Strength of Identified Source Code Properties: We use the method of 'feature importance' which quantifies how important a feature is for building a prediction model using the statistical learner, Random Forest [Cut07]. The feature importance value varies from zero to one, and a higher value for a source code property indicates higher correlation with the dependent variable. In our case, the dependent variable is the classification of a script as defective or neutral. We use Random Forests to build models using the identified properties as independent variables, and the classification of the script as defective or neutral as the dependent variable. Upon construction of the model, we compute the feature importance of each identified property provided by the Random Forest-based prediction model. To ensure stability, we follow Genuer et al. [Gen10]'s recommendations and repeat the process 10 times. We report the median feature importance values for each property, and also apply the enhanced Scott-Knott test to statistically determine which property has more feature importance, and thereby exhibits more correlation with defective scripts.

6.2.2 RQ2: Do practitioners agree with the identified source code properties?

We conduct a survey to assess if practitioners agree with the identified set of source code properties from Section 6.2.1 to be related to with defective IaC scripts. Each of the identified properties is presented as a five-point Likert-scale question. Considering the importance of a midpoint in Likert scale items [Gar91], we use a five-point scale: 'Strongly Disagree', 'Disagree', 'Neutral', 'Agree', and 'Strongly Agree'. The survey questions are available online [1].

We deployed our survey to 350 practitioners from November 2017 to July 2018. We obtained the e-mail addresses of practitioners from the collected repositories mentioned in Section 3.1.1.

6.2.3 RQ3: How can we construct defect prediction models for infrastructure as code scripts using the identified source code properties?

Figure 6.1b provides an overview of the methodology to answer RQ-3. We first apply log-transformation on the extracted counts for each source code property. The application of log transformation on numerical features helps in defect prediction and has been used in prior research [Men07a]. As described in Section 5.2.3, we apply principal component analysis (PCA) to account for multi-collinearity. We use statistical learners to construct defect prediction models, as shown in Section 2.2.2. We evaluate the constructed prediction models using 10×10 cross-validation, and four performance measures: AUC, F-Measure, precision, and recall (Section 6.2.3). We also describe the methods to which we compare our source code properties-based prediction model performance.

Comparing prediction performance: To compare prediction performance using different approaches, we use the *S*cott *K*nott (SK) test [Tan17]. SK uses hierarchical clustering analysis to partition the input data into significantly ($\alpha = 0.05$) distinct ranks [Tan17]. According to SK, an approach ranks higher if prediction performance of the constructed model using that approach is significantly higher. We use SK to compare for all four prediction performance measures: precision, recall, AUC, and F-measure.

Evaluation Methods

We use 10×10-fold cross validation to evaluate our prediction models. We use the 10×10-fold cross validation evaluation approach by randomly partitioning the dataset into 10 equal sized subsamples or folds [Tan05]. The performance of the constructed prediction models is tested by using 9 of the 10 folds as training data, and the remaining fold as test data. Similar to prior research [Gho15], we repeat the 10-fold cross validation 10 times to avoid prediction errors. We report the median prediction performance score of the 10 runs.

Comparison Model Construction

For comparison, we use two approaches: (i) the BOW approach described in Chapter 5.2, and (ii) the size-only approach where we construct defect prediction models for IaC scripts using only

[1]https://figshare.com/s/ad26e370c833e8aa9712

size of IaC scripts as measured by lines of code. In Chapter 5 we have reported that certain text features can be used to characterize defective IaC scripts and to build models to predict defective IaC scripts. Their findings are consistent with prior work in other domains, which has shown that text features are correlated with defects and good predictors of defective artifacts [Wal14]. We use the 'bag-of-words (BOW)' [Har54] technique to construct prediction models to compare our identified property-based prediction models. The BOW technique which has been extensively used in software engineering [Wal14], converts each IaC script in the dataset to a set of words or tokens, along with their frequencies. Using the frequencies of the collected tokens, we create features.

Similar to our properties derived from constructivist grounded theory, we construct defect prediction models using CART, LR, NB, and RF. We compute prediction performance using 10×10-fold cross validation, and compute precision, recall, AUC, and F-Measure. We use the Enhanced Scott-Knott test to compare if the our properties derived using the constructivist grounded theory process, significantly outperforms the text feature-based analysis, implementation smell analysis, and process metric analysis. If the text feature-based analysis is better than our derived properties, then the Scott-Knott test will rank the text feature-based analysis higher.

6.3 Results

We provide answers to the three research questions in this section.

6.3.1 RQ1: What source code properties characterize defective infrastructure as code scripts?

We use defect-related commits for the Mozilla dataset reported in Chapter 3. We use the 558 defect-related commits collected from the Mozilla dataset to identify source code properties of IaC scripts that correlate with defects. By applying constructivist grounded theory described in Section 6.2, we identify 12 properties of IaC scripts. Each of these properties are listed in Table 6.1 in the 'Property' column. A brief description of each identified property is listed in the 'Description' column.

We provide the distribution of each property for the four datasets in Table 6.2 and Figure 6.3. In Table 6.2, we report the average and maximum value for each property. Figure 6.3 presents the box plots for each property.

The median values of 12 source code properties for both defective (D) and neutral (N) scripts in presented in Table 6.3. For example, in the case of the Mirantis, the median values for 'attribute' is, respectively, 23.0 and 6.5 for defective and neutral scripts.

In Table 6.4, for each property we report the p-value and Cliff's Delta values respectively in the 'p-value' and 'Cliff' columns. We observe 10 of the 12 identified properties to show correlation with defective IaC scripts for all four datasets. The Cliff's Delta value is 'large' for lines of code for three of

Table 6.1 Source Code Properties that Characterize Defective IaC Scripts

Property	Description	Measurement Technique
Attribute	Attributes are code properties where configuration values are specified using the '=>' sign	Count of '=>' usages
Command	Commands are source code properties that are used to execute bash and batch commands	Count of 'cmd' syntax occurrences
Comment	Comments are non-executable parts of the script that are used for explanation	Count of comments
Ensure	Ensure is a source code property that is used to check the existence of a file	Count of 'ensure' syntax occurrences
File	File is a source code property used to manage files, directories, and symbolic links	Count of 'file' syntax occurrences
File mode	File mode is a source code property used to set permissions of files	Count of 'mode' syntax occurrences
Hard-coded string	Configuration values specified as hard-coded strings	Count of string occurrences
Include	Include is a source code property that is used to execute other Puppet modules and scripts	Count of 'include' syntax occurrences
Lines of code	Size of scripts as measured by lines of code can contribute to defects	Total lines of code
Require	Require is a function that is used to apply resources declared in other scripts	Count of 'require' syntax occurrences
SSH_KEY	SSH_KEY is a source code property that sets and updates ssh keys for users	Count of 'ssh_authorized_key' syntax occurrences
URL	URL refers to URLs used to specify a configuration	Count of URL occurrences

Table 6.2 Distribution of source code property values. Each tuple expresses the average and maximum count for each property for scripts.

Property	Mirantis	Mozilla	Openstack	Wikimedia
Attribute	(26.2, 249)	(11.6, 229)	(15.2, 283)	(12.6, 232)
Command	(0.3, 6)	(0.3, 5)	(0.2, 8)	(0.4, 4)
Comment	(16.2, 128)	(4.1, 121)	(33.5, 623)	(13.8, 130)
Ensure	(2.9, 42)	(1.2, 22)	(1.1, 55)	(1.5, 28)
File	(1.5, 25)	(0.7, 10)	(0.4, 15)	(1.3, 27)
File mode	(1.2, 17)	(0.6, 12)	(0.2, 11)	(0.7, 18)
Hard-coded string	(19.8, 203)	(7.1, 364)	(10.7, 212)	(9.8, 235)
Include	(5.7, 70)	(4.9, 129)	(2.6, 57)	(4.1, 29)
Lines of code	(97.5, 1287)	(52.2, 1157)	(88.2, 1222)	(58.9, 464)
Require	(2.0, 35)	(0.7, 9)	(0.5, 20)	(1.2, 11)
SSH_KEY	(1.2, 17)	(0.7, 12)	(0.2, 11)	(0.7, 18)
URL	(0.5, 9)	(1.3, 31)	(0.8, 21)	(0.4, 6)

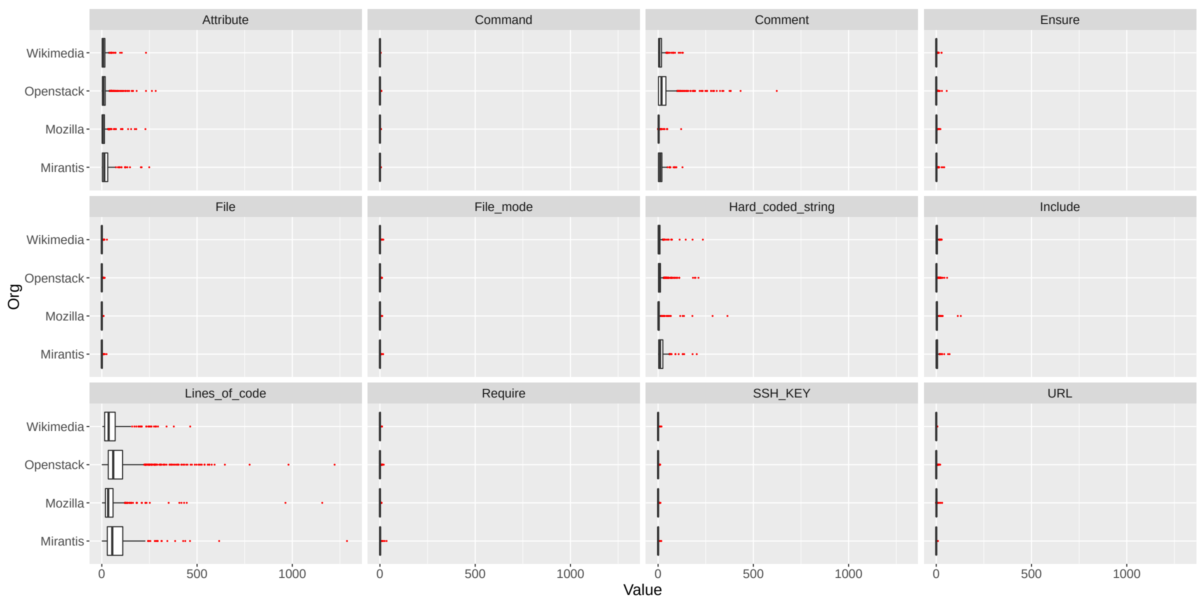


Figure 6.3 Distribution of property values for each dataset.

the four datasets. The property 'hard-coded string' has a 'large' Cliff's Delta value for Mirantis and Wikimedia.

We report the feature importance values for each identified source code property in Table 6.5. For three datasets, we observe 'lines of code' to show strongest correlation with defective scripts. For Mirantis, we observe the strongest correlation to be 'hard-coded string'.

Our findings related to feature importance is in congruence with our findings presented in Table 6.4. Cliff's delta value is 'large' for 'lines of code' for three datasets. Our feature importance analysis identifies 'lines of code' as the property with the strongest correlation for three datasets. According to Table 6.5, 'hard-coded string' is identified as the strongest correlating property and also has a 'large' Cliff's Delta value for Mirantis.

The identified source code properties that may be compatible with automated program repair tools such as, Tortoise [Wei17] are: 'attribute', 'file mode', 'file', 'hard-coded string', and 'URL'.

6.3.2 RQ2: Do practitioners agree with the identified source code properties?

As mentioned in Section 6.2.2, we conduct a survey with 350 practitioners to quantify if practitioners agreed with our set of 12 source code properties. We obtain a survey response rate of 7.4% (26 out of 350). The reported experience level in Puppet is listed in Table 6.7. The 'Experience' column lists the categories for experience in Puppet. The 'Count' column presents the number of practitioners who identified with the corresponding experience level.

Of the 12 properties, practitioners showed the highest agreement with 'include', in contrary to

Table 6.3 Median values of 12 source code properties for both defective and neutral scripts.

Property	**Mirantis**		**Mozilla**		**Openstack**		**Wikimedia**	
	D	**N**	**D**	**N**	**D**	**N**	**D**	**N**
Attribute	23.0	6.5	10.0	3.0	13.0	5.0	12.0	3.0
Comment	14.0	4.5	3.0	3.0	17.0	21.0	8.0	4.0
Command	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Ensure	1.0	1.0	1.0	1.0	0.0	0.0	1.0	0.0
File	1.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0
File mode	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Hard-coded string	19.5	4.0	4.0	2.0	8.0	4.0	8.0	2.0
Include	5.0	1.0	4.0	2.0	2.0	1.0	4.0	1.0
Lines of Code	90.0	38.0	53.0	25.0	77.0	46.0	57.0	20.0
Require	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
SSH_KEY	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
URL	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0

Table 6.4 Validation of identified source code properties. Highlighted cells in grey indicate properties for which p-value < 0.05 for all four datasets.

Property	**Mirantis**		**Mozilla**		**Openstack**		**Wikimedia**	
	p-value	**Cliff**	**p-value**	**Cliff**	**p-value**	**Cliff**	**p-value**	**Cliff**
Attribute	< 0.001	0.47	< 0.001	0.40	< 0.001	0.34	< 0.001	0.47
Command	< 0.001	0.24	< 0.001	0.18	< 0.001	0.06	< 0.001	0.18
Comment	< 0.001	0.36	0.23	0.02	0.43	0.00	< 0.001	0.22
Ensure	< 0.001	0.38	0.02	0.09	< 0.001	0.19	< 0.001	0.28
File	< 0.001	0.36	< 0.001	0.18	< 0.001	0.08	< 0.001	0.31
File mode	< 0.001	0.40	< 0.001	0.24	< 0.001	0.06	< 0.001	0.23
Hard-coded string	< 0.001	0.55	< 0.001	0.40	< 0.001	0.37	< 0.001	0.54
Include	< 0.001	0.32	< 0.001	0.31	< 0.001	0.22	< 0.001	0.37
Lines of code	< 0.001	0.50	< 0.001	0.51	< 0.001	0.32	< 0.001	0.51
Require	< 0.001	0.35	< 0.001	0.19	< 0.001	0.11	< 0.001	0.32
SSH_KEY	< 0.001	0.39	< 0.001	0.24	< 0.001	0.07	< 0.001	0.24
URL	< 0.001	0.22	0.009	0.08	0.40	0.00	< 0.001	0.17

Table 6.5 Ranked order of the 12 source code properties that show highest correlation according to feature importance analysis. The 'Practitioner Agreement' column presents the percentage of practitioners who agreed or strongly agreed with the property.

Rank	Mirantis	Mozilla	Openstack	Wikimedia	Properties Practitioners Agreed With
1	Hard-coded string (0.20)	Lines of code (0.27)	Lines of code (0.26)	Lines of code (0.19)	Include (62%)
2	Lines of code (0.17)	Attribute (0.17)	Hard-coded string (0.18)	Attribute (0.17)	Hard-coded string (58%)
3	Command (0.11)	Hard-coded string (0.15)	Attribute (0.15)	Hard-coded string (0.13)	URL (52%)
4	Comment (0.11)	Include (0.14)	Comment (0.14)	Comment (0.11)	Command (46%)
5	Attribute (0.10)	Comment (0.05)	Include (0.09)	Include (0.10)	Lines of code (42%)
6	File mode (0.08)	Ensure (0.04)	URL (0.04)	File (0.08)	Require (42%)
7	Require (0.07)	File (0.03)	Ensure (0.03)	Ensure (0.05)	File (31%)
8	Ensure (0.06)	Require (0.03)	Command (0.02)	Require (0.05)	Attribute (27%)
9	Include (0.06)	File mode (0.02)	File (0.02)	URL (0.03)	Comment (23%)
10	URL (0.03)	Command (0.02)	Require (0.02)	Command (0.02)	SSH_KEY (23%)
11	SSH_KEY (0.02)	URL (0.02)	File mode (0.01)	File mode (0.01)	Ensure (19%)
12	File (0.01)	SSH_KEY (0.01)	SSH_KEY (0.01)	SSH_KEY (0.00)	File mode (15%)

our feature importance analysis. The least agreed property is 'File mode'. Reported agreement level by all practitioners is presented in Table 6.6. For three properties we observe at least 50% of the practitioners to agree with. These three properties are: 'URL', 'hard-coded string', and 'include'.

We compare practitioner survey responses and our feature importance analysis by presenting the practitioner agreement level in Table 6.5. We also report the percentage of practitioners who agreed or strongly agreed with a certain property in the 'Practitioner Agreement' column. According to survey results, the majority of the practitioners agreed with 'include' with 61.4% agreement.

To gain further insights, we invited 20 practitioners for an interview from Dec 15, 2018 to March 25, 2019. Three practitioners responded. We asked practitioners on why they disagreed with the identified source code properties. One practitioner only agreed with 'include', stating "*'include' shows correlation with defects because these (include) often leads to errors*". The second practitioner stated that he does not hard code configuration values in his Puppet scripts, and that is why he disagreed with properties 'hard-coded string', 'file mode', and 'SSH_KEY'. He added he used the data lookup system called Hiera [2] to manage configurations. The third practitioner disagreed with 'ensure', as according to his experience this property is used to prevent defects. The practitioner agreed that properties, which are used to refer an external module or execute an external command such as 'include', 'command', and 'exec' are correlated with defects. Observations obtained from our interview analysis suggest that the disagreements are based on practitioner experience.

Despite the disagreements between our empirical findings and practitioner responses, our

[2]https://puppet.com/docs/puppet/5.4/hiera_intro.html

Table 6.6 Survey responses from practitioners. Of the 12 properties majority of the practitioners agreed with include.

	Strongly disagree (%)	**Disagree (%)**	**Neutral (%)**	**Agree (%)**	**Strongly agree (%)**
Attribute	11.8	26.9	34.6	23.0	3.8
Comment	15.6	38.4	23.0	15.3	7.7
Command	4.1	3.8	46.1	30.7	15.3
Ensure	4.1	38.4	38.4	7.7	11.53
File	4.1	15.3	50.0	23.0	7.7
File mode	4.1	19.2	61.5	11.5	3.8
Hard-coded string	3.9	11.5	26.9	46.1	11.5
Include	4.1	11.5	23.0	46.1	15.3
Lines of Code	7.9	15.3	34.6	34.6	7.7
Require	4.1	23.0	30.7	34.6	7.7
SSH_KEY	4.1	15.3	57.7	15.3	7.7
URL	8.0	11.5	30.7	46.1	3.8

Table 6.7 Reported practitioner experience in Puppet script development

Experience (Years)	**Count**
< 1	1 (3.9%)
1-2	6 (23.0%)
3-5	11 (42.3%)
6-10	7 (26.9%)
> 10	1 (3.9%)

findings can be helpful. Our findings can inform practitioners on the existence of source code properties that require sufficient inspection and testing efforts. Based on our findings, practitioners can benefit from rigorous inspection and testing when any of the 10 identified properties appear in an IaC script.

We also observe some level of congruence between our statistical analysis and survey responses. The second most agreed upon property is ‘hard-coded string’, which is identified as the most correlated property for Mirantis. So, both based on survey data and feature importance analysis, we can conclude presence of ‘hard-coded string’ in IaC scripts make scripts defect-prone.

6.3.3 RQ3: How can we construct defect prediction models for infrastructure as code scripts using the identified source code properties?

As described in Section 6.2.3, we use PCA analysis to account for collinearity before constructing the prediction models needed for RQ3. We report the number of principal components that account for at least 95% of the data variability in Table 6.8. The column ‘Property-based’ provides the number of principal components that account for 95% of the total variance where we used 12 source code properties to construct prediction models. For example, the number of principal components that account for at least 95% of the data variability for the ‘Property-based’ approach and the ‘Bag-of-words’ approach is respectively, 1 and 50.

The median AUC values are presented in Table 6.9. The column ‘Property-based’ provides the median AUC values using the 12 identified properties. For AUC the property-based prediction model outperforms the bag-of-words technique for three datasets, but is tied with the bag-of-words approach for one dataset. LR provided the highest median AUC for two datasets using our 12 properties. We report the median precision, recall, and F-measure values for 10 × 10 cross validation, for all learners and all datasets respectively in Tables 6.10, 6.11, and 6.12. The column ‘Property-based’ provides the median AUC values using the 12 identified properties, whereas, the ‘Bag-of-words’ column presents the median prediction performance values for the bag-of-words technique. As shown in Figure 6.10, for NB we observe the highest median precision for all four datasets, where the median precision is greater than 0.80. According to Table 6.11, CART provides the highest median recall for two datasets. CART and LR provides the highest median F-measure for two datasets according to Table 6.12. For three measures precision, recall, and F-measure, our property-based prediction models outperform the bag-of-words technique.

The prediction performance measures for our size-only prediction models are presented in Tables 6.13, 6.14, 6.15, and 6.16, where we respectively present the median AUC, precision, recall, and F-measure values for our size-only defect prediction models, and compare the performance with our property-based models. With respect to median AUC, size-only models outperform the property-based models for one dataset (Mirantis). For recall and F-measure, size-only models

Table 6.8 Number of Principle components used for prediction models

Dataset	Property-based	Bag-of-words
Mirantis	1	50
Mozilla	1	140
Openstack	2	400
Wikimedia	2	150

Table 6.9 AUC for each model building technique. The highlighted cell in grey indicates the best technique, as determined by the Scott Knot Test.

Dataset	Property-based				Bag-of-words			
	CART	**LR**	**NB**	**RF**	**CART**	**LR**	**NB**	**RF**
MIR	0.65	0.71	0.62	0.65	0.61	0.57	0.64	0.66
MOZ	0.71	0.69	0.66	0.69	0.52	0.51	0.60	0.56
OST	0.52	0.63	0.66	0.54	0.55	0.64	0.63	0.56
WIK	0.64	0.68	0.64	0.64	0.57	0.47	0.68	0.61

Table 6.10 Precision for each model building technique. The highlighted cell in grey indicates the best technique, as determined by the Scott Knot Test.

Dataset	Property-based				Bag-of-words			
	CART	**LR**	**NB**	**RF**	**CART**	**LR**	**NB**	**RF**
MIR	0.65	0.78	0.80	0.68	0.62	0.63	0.75	0.69
MOZ	0.68	0.73	0.85	0.67	0.51	0.48	0.39	0.58
OST	0.60	0.70	0.84	0.62	0.63	0.65	0.76	0.64
WIK	0.67	0.74	0.85	0.68	0.60	0.51	0.76	0.64

Table 6.11 Recall for each model building technique. The highlighted cell in grey indicates the best technique, as determined by the Scott Knot Test.

Dataset	Property-based				Bag-of-words			
	CART	**LR**	**NB**	**RF**	**CART**	**LR**	**NB**	**RF**
MIR	0.66	0.63	0.34	0.66	0.69	0.48	0.45	0.64
MOZ	0.66	0.54	0.37	0.64	0.25	0.21	0.39	0.27
OST	0.60	0.67	0.42	0.58	0.62	0.57	0.46	0.57
WIK	0.67	0.63	0.35	0.63	0.65	0.30	0.59	0.64

Table 6.12 F-measure for each model building technique. The highlighted cell in grey indicates the best technique, as determined by the Scott Knot Test.

Dataset	Property-based				Bag-of-words			
	CART	**LR**	**NB**	**RF**	**CART**	**LR**	**NB**	**RF**
MIR	0.67	0.70	0.48	0.67	0.66	0.64	0.63	0.67
MOZ	0.67	0.62	0.52	0.65	0.34	0.29	0.48	0.37
OST	0.60	0.68	0.56	0.60	0.62	0.61	0.58	0.60
WIK	0.67	0.68	0.50	0.66	0.63	0.38	0.66	0.65

Table 6.13 AUC for property-based models and size-only models. The highlighted cell in grey indicates the best technique, as determined by the Scott Knot Test.

Dataset	Property-based				Size-only			
	CART	LR	NB	RF	CART	LR	NB	RF
MIR	0.65	0.71	0.62	0.65	0.71	0.70	0.73	0.71
MOZ	0.71	0.69	0.66	0.69	0.67	0.65	0.62	0.67
OST	0.52	0.63	0.66	0.54	0.59	0.62	0.63	0.60
WIK	0.64	0.68	0.64	0.64	0.57	0.61	0.60	0.60

Table 6.14 Precision for property-based models and size-only models. The highlighted cell in grey indicates the best technique, as determined by the Scott Knot Test.

Dataset	Property-based				Size-only			
	CART	LR	NB	RF	CART	LR	NB	RF
MIR	0.65	0.78	0.80	0.68	0.73	0.68	0.71	0.73
MOZ	0.68	0.73	0.85	0.67	0.63	0.65	0.67	0.68
OST	0.60	0.70	0.84	0.62	0.66	0.67	0.71	0.67
WIK	0.67	0.74	0.85	0.68	0.60	0.66	0.65	0.64

Table 6.15 Recall for property-based models and size-only models. The highlighted cell in grey indicates the best technique, as determined by the Scott Knot Test.

Dataset	Property-based				Size-only			
	CART	LR	NB	RF	CART	LR	NB	RF
MIR	0.66	0.63	0.34	0.66	0.73	0.86	0.86	0.75
MOZ	0.66	0.54	0.37	0.64	0.65	0.55	0.42	0.60
OST	0.60	0.67	0.42	0.58	0.62	0.74	0.60	0.63
WIK	0.67	0.63	0.35	0.63	0.61	0.58	0.56	0.62

Table 6.16 F-measure for property-based models and size-only models. The highlighted cell in grey indicates the best technique, as determined by the Scott Knot Test.

Dataset	Property-based				Size-only			
	CART	LR	NB	RF	CART	LR	NB	RF
MIR	0.67	0.70	0.48	0.67	0.73	0.70	0.78	0.74
MOZ	0.67	0.62	0.52	0.65	0.64	0.60	0.52	0.62
OST	0.60	0.68	0.56	0.60	0.64	0.71	0.63	0.60
WIK	0.67	0.68	0.50	0.66	0.60	0.61	0.61	0.63

outperform property-based models for two datasets (Mirantis and Openstack).

Considering recall and F-measure, we observe our models constructed using size as measured by lines of code is comparable to that of the models constructed using our source code properties. However, our property-based models have higher precision, suggesting if practitioners prefer lower false positives then they can rely on the property-based defect prediction models.

Summary: In summary, we identify 10 source code properties that correlate with defective IaC scripts. Of the identified 10 properties we observe lines of code and hard-coded string i.e. putting strings as configuration values, to show the strongest correlation with defective IaC scripts. According to our survey analysis, majority of the practitioners show agreement for two properties: include, the property of executing external modules or scripts, and hard-coded string. Using the identified properties, our constructed defect prediction models show a precision of 0.70∼0.78, and a recall of 0.54∼0.67. Based on our findings, we recommend practitioners to allocate sufficient inspection and testing efforts on IaC scripts that include any of the identified 10 source code properties of IaC scripts.

CHAPTER

7 DEVELOPMENT ANTI-PATTERNS IN INFRASTRUCTURE AS CODE SCRIPTS

A development anti-pattern is a recurring development activity that relates with defective scripts, as determined by empirical analysis. We can identify development activities that relate with defective IaC scripts by mining OSS repositories. We identify development activities using metrics where a metric corresponds to a development activity. By mining metrics that correspond to a development activity, and establishing a relationship with quality of IaC scripts, we can give actionable advice for practitioners. In this chapter, we describe our study to identify development anti-patterns for IaC scripts. First, we provide related work, then we provide the methodology and the results of the study. OST

7.1 Related Work in Non-IaC Domain

This chapter is related with prior research that has studied the relationship between development activity metrics and software quality. Meneely and Williams [MW09] observed that development activity metrics, such as developer count, show a relationship with vulnerable files. Rahman and Devanbu [RD13] observed that prediction models constructed using development activity metrics are better than models built with source code metrics. Pingzer et al. [Pin08] observed that development activity metrics are related with software failures. Tufano et al. [Tuf17] used development metrics, such as developer experience, to characterize fix-inducing code changes.

Our discussion of IaC-related research described in Chapter 2.3 reveals a lack of empirical analysis that investigates what development anti-patterns may exist for IaC scripts. We address this research gap by taking inspiration from prior work that has used development activity metrics mentioned above, and apply it to the domain of IaC.

We answer the following research questions:

- RQ1: What are the development activity metrics that relate with defective infrastructure as code scripts?
- RQ2: To what extent do practitioners agree with the identified relationships between development activity metrics and defective infrastructure as code scripts? Why do they agree and disagree?
- RQ3: What development activities do practitioners suggest that relate with defective infrastructure as code scripts?

7.2 Methodology

We provide the methodology to answer the three research questions:

7.2.1 RQ1: What are the development activity metrics that relate with defective infrastructure as code scripts?

We consider seven development activity metrics: developer count, disjointness in developer groups, highest contributor's code, minor contributors, normalized commit size, scatteredness, and unfocused contribution. We consider these metrics for two reasons: (i) the metrics can provide actionable advice; and (ii) the metrics can be mined from the 94 repositories collected in Chapter 3. The criterion to determine metric actionability is "a metric has actionability if it allows a software manager to make an empirically informed decision" [Men13]. We obtain these metrics by performing a scoping review of International Conference on Software Engineering (ICSE) and Conference on Computer and Communications Security (CCS) papers published from 2008 to 2018 that are related to defect and/or vulnerability prediction. We describe the metrics below:

Developer count: Similar to prior research [Men11] [Mat10] [MW09], we hypothesize defective scripts will be modified by more developers compared to that of neutral scripts.

Disjointness in developer groups: Disjointness measures how separate one developer group is from another. A developer group is a set of developers who work on the same set of scripts [MW09]. We hypothesize that defective IaC scripts will be modified by developer groups that exhibit more disjointness compared to that of neutral scripts. Similar to prior research [MW09], with Equation 7.1

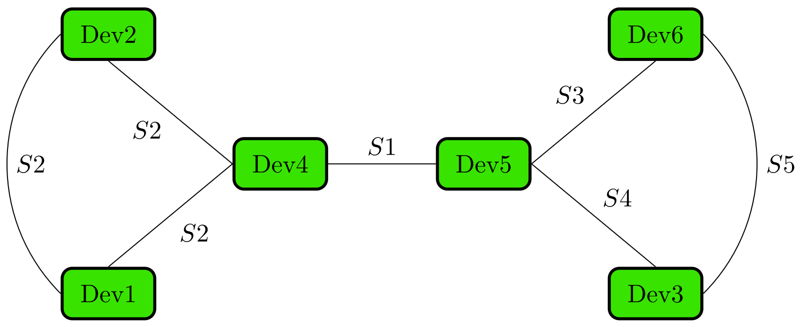


Figure 7.1 A hypothetical example to demonstrate our calculation of disjointness between developers.

we use the maximum of edge betweenness metric (MAX_EDG_BTW) in the developer network to measure the amount of disjointness between developer groups. In Equation 7.1, a and b are nodes, and e is an edge. Figure 7.1 shows a developer network. Our constructed developer network is an undirected, unweighted graph where each node corresponds to a developer. Each edge in the developer network connects two nodes if the same script is modified by developers that correspond to the two nodes of interest. In the developer network, a graph with multiple clusters indicates that developers are collaborating within themselves inside a group. These groups may work disjointly and the MAX_EDG_BTW metric can tell empirically just how disjoint are these developer groups. For example, a script modified by two developer groups with a MAX_EDG_BTW of 0.6 is likely to be more defective compared to that of a script, with MAX_EDG_BTW of 0.2.

$$MAX\_EDG\_BTW(e) = max \sum_{(a,b)\neq e} \frac{\text{\# shortest paths between a and b that pass through e}}{\text{\# shortest paths between a and b}} \tag{7.1}$$

Highest contributor's code: The highest contributor is the developer who authored the highest number of lines of code (LOC) for an IaC script. Similar to prior research [Bus17] [Bir11], we hypothesize that the highest contributor's code is significantly smaller in defective scripts compared to that of neutral scripts. We calculate the contribution by the highest contributor using Equation 7.2. The name of the metric is $HIGHEST\_CONTRIB\_CODE$.

$$HIGHEST\_CONTRIB\_CODE = \frac{\text{number of lines authored by the highest contributor}}{\text{lines of code}} \tag{7.2}$$

Minor contributors: Minor contributors is a subset of the developers who modify $\leq 5\%$ of total code for a script. Prior research [Gre15] [Bir11] reported that scripts that are modified by minor contributors are more susceptible to defects. We hypothesize that the number of minor contributors are significantly larger for defective scripts compared to that of neutral scripts.

Normalized commit size: We hypothesize that large-sized commits are more likely to appear in defective scripts than neutral scripts. We measure commit size using Equation 7.3, where we compute total lines added and deleted in all commits, and normalize by total commit count for a script.

$$Norm\_commit\_size = \frac{\sum_{i=1}^{C} \text{Total lines added/deleted in commit } i}{\text{Total number of commits}} \tag{7.3}$$

Scatteredness: Based upon findings from Hassan [Has09], we hypothesize defective scripts will include more changes that are spread throughout the script when compared to neutral scripts. In Equation 7.4, we calculate x_i, which we use in Equation 7.5 to quantify scatteredness of a script.

In our hypothetical example $Script$#1 has 10 LOC and six commits. $Script$#2 has seven LOC with four commits. For $Script$#1, three modifications are made to line#6 and 7 each. For $Script$#2, line#1, 2, 6, and 7 are modified once. According to Equation 7.5, the scatteredness score for $Script$#1 and $Script$#2 are respectively, 0.8 and 2.0.

$$x_i = \frac{\text{number of times line } i \text{ is modified}}{\text{number of commits in the script}} \tag{7.4}$$

$$Scatteredness = -\sum_{i=1}^{N} (x_i log_2 x_i) \tag{7.5}$$

Unfocused contribution: Unfocused contribution occurs when a script is changed by many developers who are also making many changes to other scripts [Pin08]. Prior research has reported unfocused contribution to be related with software failures [Pin08] and vulnerabilities [MW09]. We hypothesize that defective scripts will be modified more through unfocused contributions compared to that of neutral scripts.

Similar to prior work [MW09], we use a graph called a contribution network, which is an undirected, weighted graph. As shown in Figure 7.2, two types of nodes exist in a contribution network: script (circle) and developer (rectangle). When a developer modifies a script, an edge between that developer and that script will exist. No edges are allowed between developers or between scripts. The number of commits the developer make to a script is the weight for that edge. We use the betweenness centrality metric ($BETW\_CENT$) of the contribution network to measure unfocused contribution using Equation 7.6. Equation 7.6 presents the formula to compute betweenness centrality for script x, where a and b are developer nodes. A script with high betweenness indicates that the script has been modified by many developers who have made changes to other scripts.

$$BETW\_CENT\text{ (x)} = \sum_{a \neq x \neq b} \frac{\text{no. of shortest paths between a and b that pass through x}}{\text{no. of shortest paths between a and b}} \tag{7.6}$$

We answer RQ1 by quantifying the relationship between each metric and defective IaC scripts. For each metric, we compare the distribution for that metric between defective and neutral scripts. We use the Mann-Whitney U test [MW47] to compare the metric values for defective and neutral

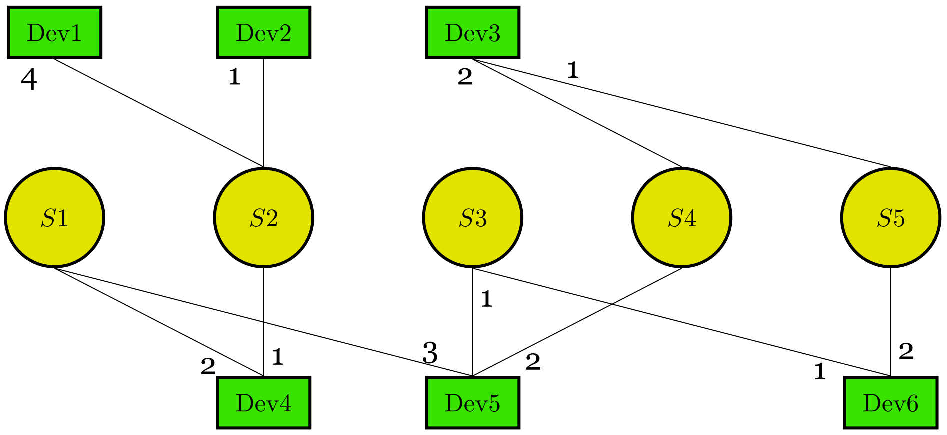


Figure 7.2 A hypothetical example to demonstrate our calculation of unfocused contribution.

scripts. The Mann-Whitney U test is non-parametric and compares two distributions and states if one distribution is significantly larger or smaller than the other. Following Cramer and Howitt's observations [CH04] to reduce Type I errors, we reject the null hypothesis if $p < 0.01$.

Along with Mann-Whitney U test, we compute effect size using Cliff's Delta, a non-parametric test [Cli93] to compare the distribution of each metric between defective and neutral scripts. The Mann-Whitney U test shows if a relationship exists, but does not reveal the magnitude of differences [SF12]. Effect size shows the magnitude of differences between two distributions. We use Romano et al. [Rom06]'s recommendations to interpret the observed Cliff's Delta values: the difference between two groups is 'large' if Cliff's Delta is greater than 0.47. A Cliff's Delta value between 0.33 and 0.47 indicates a 'medium' difference. A Cliff's Delta value between 0.14 and 0.33 indicates a 'small' difference. Finally, a Cliff's Delta value less than 0.14 indicates a 'negligible' difference.

7.2.2 RQ2: To what extent practitioners agree with the identified relationships between development activity metrics and defective infrastructure as code scripts? Why do they agree and disagree?

We answer the first part of RQ2 by deploying a survey to practitioners. In the survey, we asked the practitioners to what extent they agreed with a relationship between each of the seven metric development activity metric and defective IaC scripts. We constructed the survey following Kitchenham and Pfleeger's guidelines [KP08] as follows: (i) use Likert scale to measure agreement levels: strongly disagree, disagree, neutral, agree, and strongly agree; (ii) add explanations related to the purpose of the study, how to conduct the survey, preservation of confidentiality, relevance to participants, and an estimate of the time to complete the questionnaire; (iii) administer a pilot survey to get initial feedback on the survey. Following Smith et al. [Smi13]'s observations on software engineering survey

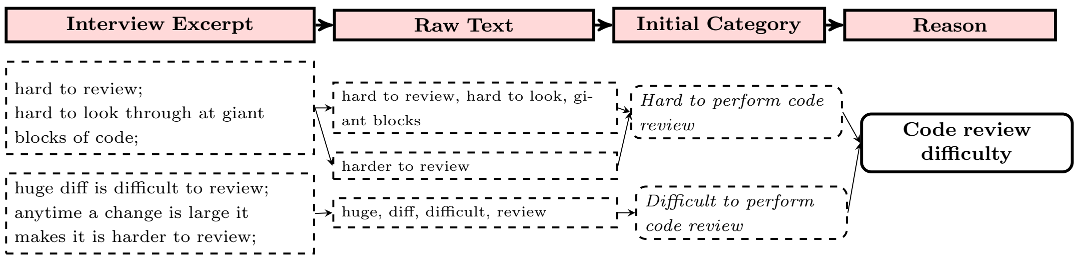

Interview Excerpt
Raw Text
Initial Category
Reason
hard to review; hard to look through at giant blocks of code;
huge diff is difficult to review; anytime a change is large it makes it is harder to review;
hard to review, hard to look, giant blocks
harder to review
huge, diff, difficult, review
Hard to perform code review
Difficult to perform code review
Code review difficulty

Figure 7.3 Example of how we use qualitative analysis to determine reasons that attribute to practitioner perception.

incentives, we offer a drawing of five Amazon gift cards as an incentive for participation.

From the pilot study feedback with five graduate students, we add an open-ended question to the survey, where we ask participants to provide any other development activity that is related with quality but not represented in our set of seven metrics. We deploy the survey to 250 practitioners from July 12, 2018 to Jan 15, 2019. We collect developer e-mail addresses from the 94 repositories. Following IRB#12598 protocol, we distribute the survey to practitioners via e-mail.

Semi-structured interview: The results from the survey indicates the level of agreement from practitioners, but not the reasons that attribute to that perception. We conducted semi-structured interviews, i.e. an interview where practitioners are asked open-ended and closed questions to identify these reasons and triangulate our findings from RQ1. We conduct all interviews over Skype or Google Hangouts. In each interview, we asked the interviewee to what extent they agree on the relationship between each metric and defective script using the Likert scale: strongly disagree, disagree, neutral, agree and strongly agree. Then, we asked about reasons that attribute to their perception. In the end, we asked interviewees if they would like to provide additional information about development activities related to defective scripts, but not included in the survey. We recruited the interviewees from the set of 250 practitioners to which we deployed the survey. We followed the guidelines provided by Hove and Anda [HA05] while conducting the interviews. We executed pilot interviews with a voluntary graduate student. During the interviews, we explained the purpose of the interview and explicitly mention that the interviewee's confidentiality will be respected.

Qualitative analysis: We transcribe the audio of each interview into textual content. From the textual content of semi-structured interviews, we apply a qualitative analysis technique called descriptive coding [Sal15] to determine reasons that can be attributed to practitioner's perceptions about the relationship between the development activity metric and defective scripts. As shown in Figure 7.3, we first identify 'raw text' from the interview. We extract raw text if any portion of the content provided a reason related to practitioner agreement or disagreement for a specific metric. For agreement, we consider agree and strongly agree. For disagreement, we consider disagree and strongly disagree. Next we generate categories from the codes. Finally, we derive the reasons from the identified categories. We report the reasons along with the corresponding metric.

7.2.3 RQ3: What development activities do practitioners suggest that relate with defective infrastructure as code scripts?

Our set of seven metrics that correspond to seven development activities is not comprehensive i.e. other development activities may exist that relate with defective IaC scripts. We use open-ended question in the survey and the semi-structured interview content to obtain development activities that are not included in our set of seven development activities. We apply qualitative analysis to identify development activities suggested by practitioners, similar to that of identifying reasons in Section 7.2.2. From the extracted content, we first identify raw text, which are abstracted into initial categories. We extract raw text if segments of the content described a development activity that is related to a defective script, but is not included in our set of seven metrics. Finally, we determine a development activity by combining initial categories based on similarities.

7.3 Results

We provide answers to the three research questions.

7.3.1 RQ1: What are the development activity metrics that relate with defective infrastructure as code scripts?

We use the Mann-Whitney U Test to compare the distribution of each metric between defective and neutral scripts and report the relationship between metrics and defective scripts. The average values of defective and neutral scripts are presented in the '(DE, NE)' column. The metrics for which $p < 0.01$, as determined by the Mann Whitney U test is indicated in grey cells. The metrics for which $p < 0.01$ for all datasets are followed by a star symbol (✪). According to Table 7.1, we observe a relationship to exist for five of the seven metrics: developer count, minor contributors, highest contributor's code, disjointness in developer groups, and unfocused contribution.

In Table 7.1, we also present the Cliff's Delta values in the Δ column. In the Δ column, we report Romano et al. [Rom06]'s interpretation of Cliff's Delta values: 'N', 'S', 'M', and 'L' respectively indicates 'negligible', 'small', 'medium', and 'large' difference. As indicated in bold, the metrics for which we observe 'medium' or 'large' differences between defective and neutral scripts across all datasets are: developer count and unfocused contribution. In short, relationship exists for five metrics, but the difference in metrics is large or medium for two: developers and unfocused contribution, which suggests that for these two metrics the differences are more observable than the other three metrics. Practitioners can use the reported effect size measures as a strategy to prioritize which development

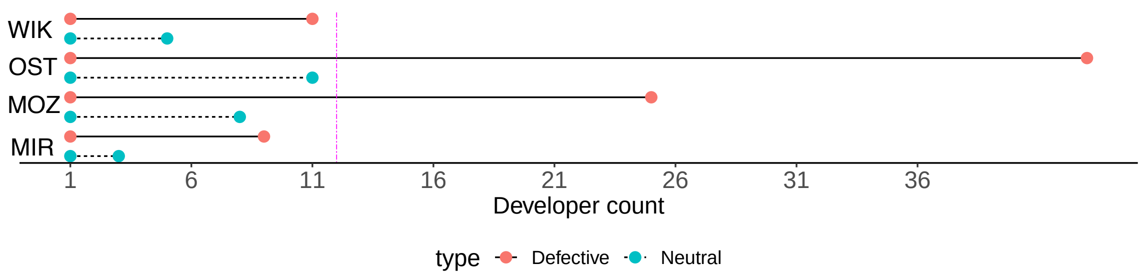


Figure 7.4 Count of developers modifying scripts. Neutral scripts are modified by no more than 11 developers.

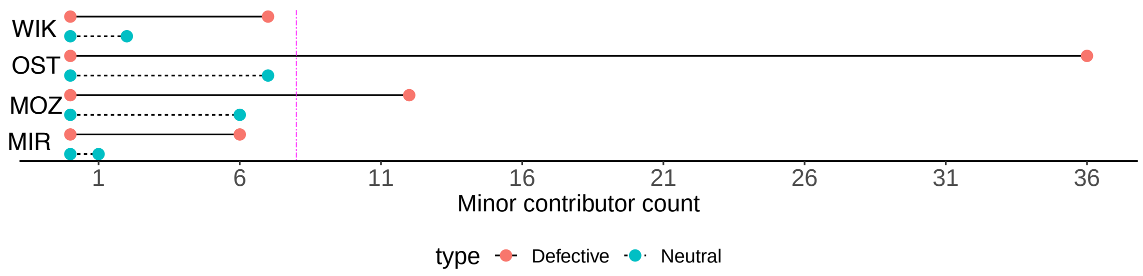


Figure 7.5 Count of minor contributors modifying scripts. Neutral scripts are modified by no more than seven minor contributors.

anti-patterns they may act upon. For example, as the difference is 'large' or 'medium' for developer count, practitioners may thoroughly inspect IaC scripts modified by multiple developers.

From Table 7.1 we observe, on average, developer count is two times higher for defective scripts than neutral scripts. Figure 7.4 presents the minimum and maximum number of developers who modify defective and neutral scripts. According to Figure 7.4, a neutral script may be modified by at most 11 developers. The dashed line shows defective scripts being modified by ≥ 12 developers, and can be modified as many as 43 developers.

From Table 7.1 we also observe, minor contributors to relate with defective scripts with a small Cliffs Delta. Figure 7.5 presents the minimum and maximum number of minor contributors who modify defective and neutral scripts. We observe a neutral script may be modified by at most seven minor contributors. The dashed line shows defective scripts being modified by ≥ 8 minor contributors, and can be modified as many as 36 minor contributors.

Table 7.1 Relationship between development activity metrics and defective IaC scripts

Metric	Mirantis		Mozilla		Openstack		Wikimedia	
	Δ	DE,NE(Avg)	Δ	DE,NE(Avg)	Δ	DE,NE(Avg)	Δ	DE,NE(Avg)
Developer count ✪	0.55 (**L**)	2.5, 1.5	0.35 (**M**)	4.1, 2.1	0.40 (**M**)	4.3, 2.2	0.36 (**M**)	2.6, 1.6
Disjointness in dev. groups ✪	0.42 (**M**)	0.4, 0.2	0.21 (S)	0.4, 0.3	0.21 (S)	0.4, 0.3	0.22 (S)	0.3, 0.2
Highest contrib. code ✪	0.41 (**M**)	0.7, 0.8	0.24 (S)	0.7, 0.8	0.37 (**M**)	0.6, 0.8	0.25 (S)	0.8, 0.9
Minor contributors ✪	0.26 (S)	0.4, 0.0	0.28 (S)	1.1, 0.2	0.30 (S)	1.6, 0.4	0.27 (S)	0.6, 0.1
Norm_commit_size	0.12 (N)	26.5, 24.1	0.13 (N)	14.0, 12.4	0.06 (N)	27.8, 28.3	0.18 (S)	18.8, 15.6
Scatteredness	0.06 (N)	2.6, 2.6	0.38 (**M**)	2.9, 2.2	0.15 (S)	3.4, 3.3	0.32 (S)	3.0, 2.4
Unfocused contribution ✪	0.56 (**L**)	2.5, 1.5	0.35 (**M**)	3.4, 1.8	0.40 (**M**)	4.3, 2.2	0.36 (**M**)	2.6, 1.6

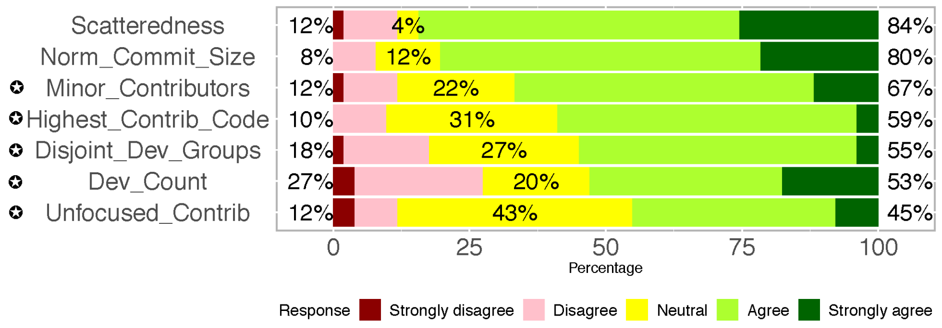


Figure 7.6 Survey findings that illustrate to what extent practitioners agree with the seven metrics and their relationships with defective scripts.

7.3.2 RQ2: To what extent do practitioners agree with the identified relationships between development activity metrics and defective infrastructure as code scripts? Why do they agree and disagree?

We obtained a 20.4% response rate for our survey, which is typical for software engineering-based surveys [Smi13]. The average experience in Puppet of the survey respondents is 3.9 years. We present the findings in Figure 7.6. The star symbol represents the metrics that relate with defective scripts identified from our quantitative analysis stated in Section 7.3.1. The percentage values on the right hand side correspond the percentage of practitioners to which agree or strongly agree with. For example, for scatteredness we observe 84% of practitioners to agree or strongly agree with. At least 80% of the practitioners who responded, agreed or strongly agreed that scatteredness and normalized commit size and are related with quality of IaC scripts. The least agreed upon metric is unfocused contribution with 45% agreement.

Of the 51 survey respondents, 11 agreed to participate for semi-structured interviews. Of the 11 interviewees, eight were developers, two were DevOps consultants, and one was a project manager. The average experience of the 11 interviewees is 5.1 years. The duration of interviews varied from 13 to 29 minutes. Through our qualitative analysis of interview content, we identified a set of reasons that attribute to the agreements or disagreements. For the seven metrics, the author of this dissertation and a colleague, respectively, obtained 8 and 10 reasons that attribute to practitioner perception. The Cohen's Kappa is 0.87, and we resolve the disagreements for the identified reasons by discussing if additional reasons identified by the colleague are similar to the existing reasons that the dissertation author and the colleague have identified. Upon resolving disagreements, we obtain eight reasons that are identified both by the dissertation author and the colleague.

Why do they agree and disagree?: The reasons for agreement and disagreement are respectively underlined using a solid and dashed line. The number of interviewees who agree and disagree are enclosed in a parenthesis.

Agreement and disagreement reasons for five metrics related to defective scripts are stated below:

Developer count (8, 3): Developer count and defective scripts are related because of communication problems, as the complexity of maintaining proper communication may increase. One interviewee stated"*obviously the more people have their hand on the code base, and if good communication isn't happening, then they're gonna have different opinions about how things should end up*". Brooks [Bro95] provided a rule of thumb stating"*if there are n workers working on a project...there are potentially almost 2^n teams within which coordination must occur*", indicating adding developers to software development will result in increased communication complexity. Disagreeing interviewees stated when developers work on the same script knowledge sharing happens, which can increase the quality of IaC scripts. One interviewee also reminded about reality:"*I don't think its reasonable to expect that each script is maintained by only a low amount of people. Generally, they [teams] strive to have many people to write as many scripts as possible because that one guy who understands that code can leave*".

Disjointness (7, 3): Disjointness between developer groups and defective scripts can be related because of communication problems between the groups. Developers each have their own opinions, and without coordination, scripts are likely to be defective. According to disagreeing practitioners, due to knowledge sharing, collaboration between developers, even if in disjoint groups, will lead to increased quality of IaC scripts because collaboration leads to knowledge sharing.

Minor contributors (8, 3): The relationship between minor contributors and defective scripts can exist because of lack of context: developers who contribute less do not have the full context compared with the developers who contribute the majority of the code. Interviewees disagreed for knowledge sharing. Developers regardless of their contribution amount, working on the same script can share the common knowledge needed to modify the script mitigating defects.

Highest contributor's code (8, 3): Interviewees mentioned lack of context: other developers do not have the full the context as the highest contributor, and if the other developers contribute more than the highest contributor, the script suffers from quality issues. On the contrary, due to knowledge sharing, the highest contributor's code amount may not relate with defective scripts.

The highest contributor can share knowledge with other developers on how to modify the script without introducing defects.

Unfocused contribution (7, 4): Distraction is why unfocused contribution is related to defective scripts. Working on multiple scripts can lead to distraction, which could influence the quality of IaC script development. Their reasoning is congruent with Weinberg [Wei92], who proposed a rule of thumb suggesting a 10% decrease in software quality whenever developers switch between projects. Interviewees who disagreed stated the bubble problem–developers may get stuck in a 'bubble' if they don't work on multiple scripts. When developers work on multiple scripts, they gather knowledge collected from one script and apply it to another script. One interviewee said "*working on one script means you are in a bubble, you cannot see what solutions were conceived to solve certain problems*".

Agreement and disagreement reasons for two metrics not related to defective scripts are stated below:

Normalized commit size (8, 2): Contrary to our quantitative findings, interviewees stated commit size and defective scripts is related for two reasons: code review difficulty and debugging difficulty. Researchers [Mac18] have reported large-sized commits are harder to review, and susceptible of introducing defects. Their observation are also supported by anecdotal evidence by industry experts in IaC [Mor16], who advise in committing small changes at a time. Furthermore, with large-sized commits, bug localization becomes challenging. One interviewee mentioned"*you should always split your changes ... this [large-sized] commit is not helping you to find when and where the bug was introduced; it is harder to fix and revert*". Reviewing Puppet code can be harder according to one interviewee"*reviewing Puppet code is more challenging than regular code ... if I look for a JavaScript diff I can look side by side and see what the new functions do, whereas with Puppet you have to think it through your head*". Interviewees also mentioned how large-sized commits are introduced. One interviewee mentioned IaC scripts can be authored by system operators who do not have a software development background:"*Ansible and Puppet modules are written by sysadmins and not developers...we have been in this shift where sysadmins become infrastructure developers and the setup of being an admin to a developer is different because there is a difference between a developer and a sysadmin.*". Disagreeing interviewees mentioned developer experience. A developer who is well-versed with the script may be able to submit a large-sized commit without making the script defective.

Scatteredness (7, 3): Code review difficulty is why interviewees perceive scatteredness is related with defective IaC scripts. If the changes are spread throughout the script, developers may miss a defect while reviewing the code changes. On the contrary, three interviewees disagreed stating lack of context as the reason.

7.3.3 RQ3: What development activities do practitioners suggest that relate with defective infrastructure as code scripts?

From the interview excerpts, the dissertation author and colleague, respectively, obtained 8 and 10 development anti-patterns, where the Cohen's Kappa is 0.71. The dissertation author and colleague, respectively, obtained five and four development anti-patterns by analyzing the open-ended text from the survey with a Cohen's Kappa of 0.85. We resolve disagreements through discussions in both cases. From the interview excerpt and the open-ended question in the survey we identify seven development activities related to defective scripts, and present them alphabetically below:

Change management process: Some IT organizations do not have a change management process. Changes in IaC scripts should be managed systematically, where the scripts are maintained in version control, and changes in IaC scripts must be checked in regularly. Changes should also be peer-reviewed so that defects do not slip through in the production severs. Humble and Farley stated adequate change management process to be vital for IaC [HF10], as a small change can lead to erroneous server configurations.

Design: IaC scripts can be developed on an ad-hoc basis without following any specific design"*Usually the approach is to 'get it working' with little focus on design*". An adequate design can guide how scripts need to be implemented, as one survey responded stated "*Much more important is that fact, that you need to have a good design phase in advance, so that you know which pieces need to be implemented*".

Documentation: Practitioners emphasized that documentation was necessary to increase quality of IaC scripts. One survey respondent mentioned "*I see a lot of undocumented code where the information why something is done is missing*", which refers to source code comments that do not explain why the source code is added. One interviewee stated"*unlike traditional programming languages C++/Java, people working with Puppet tend not to comment the code*".

Feature flags: Feature flag is the technique that facilitates in triggering a specific portion of the software code [Fow17]. Practitioners recommend the use of feature flags as it allows practitioners to control which parts of an IaC script can be executed. If changes are scattered or included in a large-sized commit, then practitioners use feature flags to test if the changes are correct. In this manner, defects can be prevented in IaC scripts.

Gradual rollouts: Gradual rollout is the step-by-step process of deploying software changes to fractions of end-users [HF10]. According to surveyed practitioners with gradual rollout, developers can get an early assessment on the quality of their submitted code changes and use the feedback to improve the quality of IaC scripts and to reduce the propagation of wide-scale outages.

Inspection efforts: Due to lack of validation and verification resources one interviewee stated that not all scripts are review-able, and scripts that are more likely to be defective needs to be separated and prioritized for inspection. Another interviewee suggested prioritizing scripts that are likely to be defect-prone using machine learning "*I believe that machine learning will be very useful in identifying buggy code*".

Testing: Interviewees mentioned testing is not done often for IaC scripts. We identify three types of testing from interview excerpts: functional, unit, and acceptance. IaC scripts that undergo sufficient testing also increases developer confidence, as one interviewee expressed "*I felt more comfortable when the code went through a lot of testing*".

Summary: We identify a set of development anti-patterns based on two criteria: each activity must be (i) related to software quality as reported in prior work and (ii) supported by quantitative evidence across all four datasets (Section 7.3.1) or by qualitative analysis (Section 7.3.2). We alphabetically list the identified 12 development anti-patterns with definitions, and possible solutions for each anti-pattern.

Anti-pattern#1: Boss is Not Around
Definition: The highest contributor is not completely involved in the authorship of an IaC script.
Observation: The highest contributor may have the full context of the script, which other developers may not have. If the other developers contribute more than the highest contributor, the corresponding script may be defective. On average, the highest contributor contributes 80%∼90% of the code for neutral scripts.
Source: Quantitative analysis (Metric: Highest contributor's code)
Solution: Perform team planning so that the highest contributor can modify and/or verify all the content added to a script.

Anti-pattern#2: Code Without Design
Definition: Developing IaC scripts without specifying a design.
Observation: Developers often write IaC scripts without following a specific design.
Source: Interview
Solution: Perform design before developing scripts.

Anti-pattern#3: Inadequate Change Management Process
Definition: Not using a change management process for IaC.
Observation: A systematic change management process for IaC is essential, which includes version control, automated and/or manual code reviews.
Source: Interview
Solution: Include version control and code review as part for the change management process for IaC.

Anti-pattern#4: Inadequate Documentation

Definition: The activity of not properly documenting IaC scripts through source code comments.
Observation: Unlike Java code, IaC scripts are not well-documented e.g. not adding comments.
Source: Interview
Solution: Put comments in IaC scripts describing the purpose and expected output.

Anti-pattern#5: Inefficient Allocation of Inspection Efforts
Definition: Not prioritizing scripts for inspection efforts.
Observation: IT organizations may have to maintain large number of IaC scripts and for sufficient validation scripts need to be prioritized.
Source: Interview
Solution: Prioritizing scripts based on defect-proneness possibly using machine learning.

Anti-pattern#6: Lack of Testing
Definition: The activity of not testing IaC scripts.
Observation: Testing is not done frequently. Inadequate use of unit/functional/acceptance testing for IaC scripts.
Source: Interview
Solution: Rely on tools such as rspec-puppet [**rspec:puppet**] for testing scripts.

Anti-pattern#7: Many Cooks Spoil
Definition: Having multiple developers working on the same script.
Observation: Defective scripts are modified by 12∼43 developers, whereas, neutral scripts modified by no more than 11 developers.
Source: Quantitative analysis (Metric: Developer count)
Solution: Thoroughly inspect scripts modified by multiple developers.

Anti-pattern#8: Minors are Spoilers
Definition: The activity of a script being modified by developer(s) who writes no more than 5% code of the script.
Observation: A neutral script may be modified by at most seven minor contributors, whereas, defective scripts can be modified by 8∼36 minor contributors.
Source: Quantitative analysis (Metric: Minor contributors)
Solution: Thoroughly inspect scripts modified by multiple minor contributors.

Anti-pattern#9: Not Using Gradual Rollouts

Definition: The activity of not using gradual rollouts.
Observation: Without the use of gradual rollout defects can be introduced in IaC scripts. Gradual rollout helps in obtaining early feedback on the submitted code changes.
Source: Interview
Solution: Use gradual rollouts to obtain early feedback.

Anti-pattern#10: Not Using Feature Flags
Definition: The activity of not using feature flags.
Observation: Without the use of feature flags practitioners may not be able to mitigate defects that may arise from scatteredness.
Source: Interview
Solution: Use feature flags to mitigate defects.

Anti-pattern#11: Silos
Definition: The activity of developers working in disjoint groups.
Observation: Defective scripts are modified by developer groups, which are 1.3∼2.0 times disjoint, on average, compared to that of neutral scripts. Even though practitioners perceive IaC as a tool to break silos [Bri17], our quantitative findings suggest that silos exist in IaC development.
Source: Quantitative analysis (Metric: Disjointness in developer groups)
Solution: Collaborate more instead of working in disjoint groups in IaC development.

Anti-pattern#12: Unfocused Contribution
Definition: The activity of developers working on an IaC script, who also modify other scripts.
Observation: On average, for defective scripts unfocused contribution is 62.5%∼95.4% higher than neutral scripts.
Source: Quantitative analysis (Metric: Unfocused contribution)
Solution: Perform team planning so that developers can focus on developing one script development within a certain time period.

CHAPTER

8 SECURITY ANTI-PATTERNS

This chapter focuses on identifying security anti-patterns existing in IaC scripts. Security anti-patterns are recurring coding patterns that are indicative of security weakness, and warranting further inspection. A code anti-pattern is a recurrent coding pattern that is indicative of potential maintenance problems [FB99]. A code anti-pattern may not always have bad consequences, but still deserves attention, as a code anti-pattern may be an indicator of a problem [FB99]. Security anti-patterns are different from vulnerabilities, because a vulnerability is a weakness in software, which upon exploitation results in a negative impact [Nat18]. The existence and persistence of security smells in IaC scripts leave the possibility of another programmer using these smelly scripts, potentially propagating use of insecure coding practices. We organize this chapter by first providing related work. Next, we provide methodology. Finally, we provide results of the study.

8.1 Related Work in Non-IaC Domain

Prior research has investigated what categories of bad coding practices can have security consequences in non-IaC domains, such as Android applications and Java frameworks. Meng et al. [Men18] studied bad coding practices related to the security of Java Spring Framework in Stack Overflow, and reported 9 out of 10 SSL/TLS-related posts to discuss insecure coding practices. Rahman et al. [Rah19a] studied Python code blocks posted on Stack Overflow, and observed that 7.1% of the 44,966 Python-related answers to include at least one insecure coding practice. Fahl et al. [Fah12] investigated inappropriate use of SSL/TLS protocols, such as, trusting all certificates and stripping SSL, for Android applications. Using MalloDroid, Fahl et al. [Fah12] identified 8% of the studied 13,500 Android applications to inappropriately use SSL/TLS. Felt et al. [Fel11] used Stowaway to study if Android applications follow the principle of least privilege. They reported 323 of the studied 900 Android applications to be over-privileged. Ghafari et al. [Gha17] analyzed 70,000 Android applications to identify the frequency of security smells. They reported 50% of the studied applications to contain at least three security smells. Egele et al. [Ege13] used a static analysis tool called Cryptolint to analyze if cryptography APIs are used inappropriately, for example, using constant encryption keys and using static seeds to seed pseudo-random generator functions. Egele et al. [Ege13] observed at least one inappropriate use for 88% of 11,748 Android applications.

The above-mentioned work highlights that other domains such as Android are susceptible to inappropriate coding practices that have security consequences. We take motivation from this prior work and hypothesize that security-related anti-patterns exist also for IaC scripts. We evaluate our hypothesis by conducting an empirical study on identifying security anti-pattern in IaC scripts.

We answer the following research questions:

- RQ1: What security anti-patterns appear in infrastructure as code scripts?
- RQ2: How frequently do security anti-patterns occur?
- RQ3: How do practitioners perceive the identified security anti-patterns?

8.2 Methodology

We answer RQ1 using open coding described below. Next, we answer RQ2 by constructing a static analysis tool called Security Linter for Infrastructure as Code (SLIC).

8.2.1 RQ1: What security anti-patterns occur in infrastructure as code scripts?

Data Collection: We collect 1,726 Puppet scripts from the Mozilla and Openstack datasets mentioned in Chapter 3 to determine the security anti-patterns. We collect these scripts from the master branches of 74 repositories, downloaded on July 30, 2017. We collect these 74 repositories from the three organizations Mozilla, Openstack and Wikimedia Commons. We use Puppet scripts to construct our dataset because Puppet is considered as one of the most popular tools for configuration management [JA15] [Sha16a], and has been used by companies since 2005 [MJ11].

Qualitative Analysis: We first apply a qualitative analysis technique called descriptive coding [Sal15] on 1,726 Puppet scripts to identify security anti-patterns. Next, we map each identified anti-pattern to a possible security weakness defined by CWE [MIT18]. We select qualitative analysis because we can (i) get a summarized overview of recurring coding patterns that are indicative

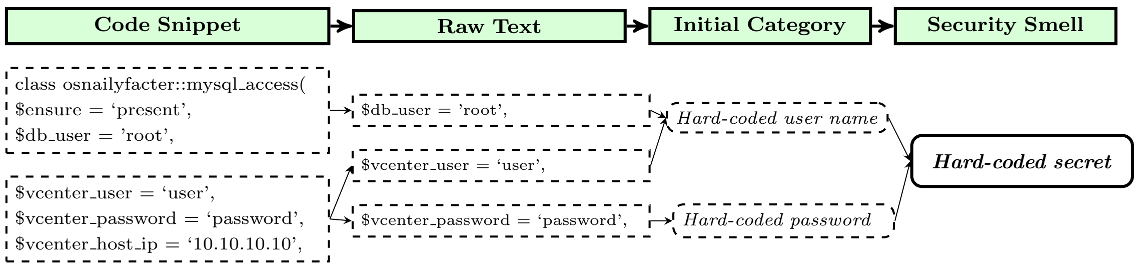


Figure 8.1 An example of how we use qualitative analysis to determine security smells in Puppet scripts.

of security weakness; and (ii) obtain context on how the identified security anti-patterns can be automatically identified. We determine security anti-patterns by first identifying code snippets that may have security weaknesses based on the dissertation author’s security expertise. Next, we search the CWE database to validate the identified security anti-pattern. We select the CWE to map each anti-pattern to a security weakness because CWE is a list of common software security weaknesses developed by the security community [MIT18].

Figure 8.1 provides an example of our qualitative analysis process. We first analyze the code content for each IaC script and extract code snippets that correspond to a security weakness as shown in Figure 8.1. From the code snippet provided in the top left corner, we extract the raw text: ‘$db_user = ‘root’. Next we generate the initial category ‘Hard-coded user name’ from the raw text ‘$db_user = ‘root’’ and ‘$vcenter_user = ‘user’’. Finally, we determine the anti-pattern ‘Hard-coded secret’ by combining initial categories. We combine these two initial categories, as both correspond to a common pattern of specifying user names and passwords as hard-coded secrets. Upon derivation we observe ‘Hard-coded secret’ to be related to ‘CWE-798: Use of Hard-coded Credentials’ and ‘CWE-259: Use of Hard-coded Password’ [MIT18].

Security Linter for Infrastructure as Code Scripts (SLIC): We first describe how we construct SLIC, then we describe how we evaluate SLIC’s smell detection accuracy.

```
1  #This is an example Ansible script
2  – name: install docker
3  – package:
4      name: python3
5      gpgcheck: false
```

(a)

Line#	Output of Parser
1	<COMMENT, ‘This is an example Ansible script’>
2	<KEY, ‘name’, ‘install docker’>
3	<KEY, ‘package’, ‘{‘name’, ‘gpgcheck’}’>
4	<KEY, ‘name’, ‘python3’>
5	<KEY, ‘gpgcheck’, ‘false’>

(b)

Figure 8.2 Output of the ‘Parser’ component in SLIC for Ansible. Figure 8.4a presents an example Ansible script fed to Parser. Figure 8.4b presents the output of Parser for the example Ansible script.

```
#This is an example Chef script
tempVar = 1
file "/tmp/test.txt" do
  content "Test file."
  owner "test"
  group "test"
  mode "00600"
end
```

(a)

Line#	Output of Parser
1	<COMMENT, 'This is an example Chef script'>
2	<VARIABLE, 'tempVar', '/tmp/test.txt'>
3	<RESOURCE, 'file', '/tmp/test.txt'>
4	<PROPERTY, 'content', 'Test file'>
5	<PROPERTY, 'owner', 'test'>
6	<PROPERTY, 'group', 'test'>
7	<PROPERTY, 'mode, '00600'>

(b)

Figure 8.3 Output of the 'Parser' component in SLIC for Chef. Figure 8.4a presents an example Chef script fed to Parser. Figure 8.4b presents the output of Parser for the example Chef script.

8.2.2 Description of SLIC

SLIC is a static analysis tool for detecting security smells in IaC scripts. SLIC has two components:

Parser: The Parser parses an IaC script and returns a set of tokens. Tokens are non-whitespace character sequences extracted from IaC scripts, such as keywords and variables. Except for comments, each token is marked with its name, token type, and any associated configuration value. Only token type and configuration value are marked for comments. For example, Figures 8.2a and 8.4a respectively provides a sample script in Ansible and Chef that is fed into SLIC. The output of Parser is expressed as a vector, as shown in Figures 8.2b and 8.4b. For example, the comment in line#1, is expressed as the vector '<COMMENT, 'This is an example Chef script'>'.

In the case of Ansible, Parser first identifies comments. Next, for non-commented lines Parser uses a YAML parser and constructs a nested list of key-values pairs in JSON format. We use these key-value pairs to construct rules for the Rule Engine.

In the case of Puppet, except for comments, each token is marked with its name, token type, and any associated configuration value. Only token type and configuration value are marked for comments. For example, Figure 8.4a provides a sample script that is fed into SLIC. The output of Parser is expressed as a vector, as shown in Figure 8.4b. For example, the comment in line#1, is expressed as the vector '<COMMENT, 'This is an example Puppet script'>'. Parser provides a vector representation of all code snippets in a script.

Rule Engine: We take motivation from prior work [Tan07] and use a rule-based approach to detect security smells. We use rules because (i) unlike keyword-based searching, rules are less susceptible to false positives [Tan07]; and (ii) rules can be applicable for IaC tools irrespective of their syntax. The Rule Engine consists of a set of rules that correspond to the set of security smells identified in Section 8.2. The Rule Engine uses the set of tokens extracted by Parser and checks if any rules are satisfied.

We can abstract patterns from the smell-related code snippets, and constitute a rule from the generated patterns. We use Table 8.2 to demonstrate our approach. The 'Code Snippet' column presents a list of code snippets related to 'Use of HTTP without TLS'. The 'Parser Output' column represents vectors for each code snippet. We observe that the vector of format '<VARIABLE, NAME, CONFIGURATION VALUE >' and '<PROPERTY, NAME, CONFIGURATION VALUE >', respectively, occurs three times and twice for our example set of code snippets. We use the vectors from the output of 'Parser' to determine that variable and properties are related to 'Use of HTTP without TLS'. The vectors can be abstracted to construct the following rule: '$(isVariable(x) \vee isProperty(x))$ $\wedge$ $isHTTP(x)$'. This rule states that 'for an IaC script, if token x is a variable or a property, and a string is passed as configuration value for a variable or a property which is related to specifying a URL that uses HTTP without TLS support, then the script contains the security smell 'Use of HTTP without TLS'. We apply the process of abstracting patterns from smell-related code snippets to determine the rules for the all security smells for both Ansible and Chef.

A programmer can use SLIC to identify security smells for one or multiple Puppet scripts. The programmer specifies a directory where script(s) reside. Upon completion of analysis, SLIC generates a comma separated value (CSV) file where the count of security smell for each script is reported. We implement SLIC using API methods provided by PyYAML [1] for Ansible and Foocritic [2] for Chef.

Rules to Detect Security Smells: For Ansible and Chef we present the rules needed for the 'Rule Engine' of SLIC respectively in Tables 8.4 and 8.5. The string patterns need to support the rules in Tables 8.4 and 8.5 are listed in Table 8.6. The 'Rule' column lists rules for each smell that is executed by Rule Engine to detect smell occurrences. To detect whether or not a token type is a resource ($isResource(x)$), a property ($isProperty(x)$), or a comment ($isComment(x)$), we use the token vectors generated by Parser. Each rule includes functions whose execution is dependent on matching of string patterns. We apply a string pattern-based matching strategy similar to prior work [Bos14], where we check if the value satisfies the necessary condition. Table 8.6 lists the functions and corresponding string patterns. For example, function 'hasBugInfo()' will return true if the string pattern 'show_bug\.cgi?id=[0-9]+' or 'bug[#$\backslash t$]$*$[0-9]+' is satisfied.

For Ansible, the rule engine takes output from the Parser, which indicates if each processed line is a comment or a key-value pair. For example, according to Table 8.4, for the key value pair (k, $k.value$), the rule engine detects an empty password if the key k includes a string indicative of a password ($isPassword$ in Table 8.6), and the length of the value ($k.value$) is equal to zero.

[1]https://pyyaml.org/
[2]http://www.foodcritic.io/

Table 8.1 An Example of Using Puppet Code Snippets To Determine Rule for 'Hard-coded secret'

Code Snippets	Output of Parser
$keystone_db_password = 'keystone_pass',	<VARIABLE, '$keystone_db_password', 'keystone_-pass' >
$glance_user_password = 'glance_pass',	<VARIABLE, '$glance_user_password', 'glance_pass' >
$rabbit_password = 'rabbit_pw',	<VARIABLE, '$rabbit_password', 'rabbit_pw' >
user =>'jenkins'	<ATTRIBUTE, 'user', 'jenkins' >
$ssl_key_file = '/etc/ssl/private/ssl-cert-gerrit-review.key'	<VARIABLE, '$ssl_key_file', '/etc/ssl/private/ssl-cert-gerrit-review.key' >

Table 8.2 An Example of Using Chef Code Snippets To Determine Rule for 'Use of HTTP Without TLS'

Code Snippets	Output of Parser
repo='http://ppa.launchpad.net/chris-lea/node.js-legacy/ubuntu'	<VARIABLE, 'repo', 'http://ppa.launchpad.net/chris-lea/node.js-legacy/ubuntu' >
repo='http://ppa.launchpad.net/chris-lea/node.js/ubuntu'	<VARIABLE, 'repo', 'http://ppa.launchpad.net/chris-lea/node.js/ubuntu' >
auth_uri='http://localhost:5000/v2.0'	<VARIABLE, 'auth_uri', 'http://localhost:5000/v2.0' >
uri 'http://binaries.erlang-solutions.com/debian'	<PROPERTY, 'uri', 'http://binaries.erlang-solutions.com/debian' >
url 'http://pkg.cloudflare.com'	<PROPERTY, 'url', 'http://pkg.cloudflare.com' >

8.2.3 Evaluation of SLIC

Any static analysis tool is subject to evaluation. We use raters to construct the oracle dataset to mitigate the dissertation author's bias in SLIC's evaluation, similar to Chen et al. [CJ17].

Oracle Dataset: For Ansible and Chef we construct two oracle datasets using closed coding [Sal15], where a rater identifies a pre-determined pattern. For the Ansible oracle dataset, 96 scripts are manually checked for security smells by at least two raters. For the Chef oracle dataset, 76 scripts are manually checked for security smells by at least two raters. The raters apply their knowledge related to IaC scripts and security, and determine if a certain smell appears for a script. For both oracle datasets we did not include any raters as part of deriving smells or constructing SLIC to avoid bias.

```
1 #This is an example Puppet script
2 $token = `XXXYYYZZZ'
3 $os_name = `Windows'
4 auth_protocol => `http'
5 $vcenter_password = `password'
```

(a)

Line#	Output of Parser
1	<COMMENT, 'This is an example Puppet script'>
2	<VARIABLE, 'token', 'XXXYYYZZZ'>
3	<VARIABLE, 'os_name', 'Windows'>
4	<ATTRIBUTE, 'auth_protocol', 'http'>
5	<VARIABLE, 'vcenter_password', 'password'>

(b)

Figure 8.4 Output of the 'Parser' component in SLIC for Puppet. Figure 8.4a presents an example Puppet script fed to Parser. Figure 8.4b presents the output of Parser for the example Puppet script.

Table 8.3 Rules to Detect Security Smells for Puppet Scripts

Smell Name	Rule
Admin by default	$(isParameter(x)) \wedge (isAdmin(x.name) \wedge isUser(x.name))$
Empty password	$(isAttribute(x) \vee isVariable(x)) \wedge ((length(x.value) == 0 \wedge isPassword(x.name))$
Hard-coded secret	$(isAttribute(x) \vee isVariable(x)) \wedge (isUser(x.name) \vee isPassword(x.name) \vee isPvtKey(x.name)) \wedge (length(x.value) > 0)$
Invalid IP address binding	$((isVariable(x) \vee isAttribute(x)) \wedge (isInvalidBind(x.value))$
Suspicious comment	$(isComment(x)) \wedge (hasWrongWord(x) \vee hasBugInfo(x))$
Use of HTTP without TLS	$(isAttribute(x) \vee isVariable(x)) \wedge (isHTTP(x.value))$
Use of weak crypto. algo.	$(isFunction(x) \wedge usesWeakAlgo(x.name))$

Table 8.4 Rules to Detect Security Smells for Ansible Scripts

Smell Name	Rule
Empty Password	$iskey(k) \wedge length(k.value) == 0 \wedge isPassword(k)$
Hard-coded Secret	$((isKey(k) \wedge length(k.value) > 0) \wedge (isUser(k) \vee isPassword(k) \vee isPrivateKey(k)))$
Invalid IP Address Binding	$(isKey(k) \wedge isInvalidBind(k.value))$
No integrity check	$(isKey(k) \wedge (isIntegrityCheck(x) == False \wedge isDownload(x.value)))$
Suspicious Comment	$isComment(k) \wedge hasWrongWord(k) \wedge hasBugInfo(k)$
Use of HTTP without TLS	$isKey(k) \wedge isHTTP(k.value)$

Table 8.5 Rules to Detect Security Smells for Chef Scripts

Smell Name	Rule
Admin by default	$(isPropertyOfDefaultAttribute(x)) \wedge (isAdmin(x.name) \wedge isUser(x.name))$
Empty password	$(isProperty(x) \vee isVariable(x)) \wedge ((length(x.value) == 0 \wedge isPassword(x.name))$
Hard-coded secret	$(isProperty(x) \vee isVariable(x)) \wedge (isUser(x.name) \vee isPassword(x.name) \vee isPvtKey(x.name)) \wedge (length(x.value) > 0)$
Invalid IP address binding	$((isVariable(x) \vee isProperty(x)) \wedge (isInvalidBind(x.value))$
Missing default in case	$(isCaseStmt(x) \wedge x.elseBranch == False)$
No integrity check	$(isProperty(x) \vee isAttribute(x)) \wedge (isIntegrityCheck(x) == False \wedge isDownload(x.value))$
Suspicious comment	$(isComment(x)) \wedge (hasWrongWord(x) \vee hasBugInfo(x))$
Use of HTTP without TLS	$isProperty(x) \vee isVariable(x)) \wedge (isHTTP(x.value)$
Use of weak crypto. algo.	$isAttribute(x) \wedge usesWeakAlgo(x.value)$

Table 8.6 String Patterns Used for Functions in Rules

Function	**String Pattern**
$hasBugInfo()$ [Sli05]	‘bug[#\ t]∗[0-9]+’,‘show_bug\.cgi?id=[0-9]+’
$hasWrongWord()$ [MIT18]	‘bug’, ‘hack’, ‘fixme’, ‘later’, ‘later2’, ‘todo’
$isAdmin()$	‘admin’
$isDownload()$	‘http[s]?://(?:[a-zA-Z]||[0-9]||[$-_@.&+]||[!*,]|(?:%[0-9a-fA-F][0-9a-fA-F]))+.[dmg|rpm|tar.gz|tgz]’
$isHTTP()$	‘http:’
$isInvalidBind()$	‘0.0.0.0’
$isIntegrityCheck()$	‘gpgcheck’, ‘check_sha’, ‘checksum’, ‘checksha’
$isPassword()$	‘pwd’, ‘pass’, ‘password’
$isPvtKey()$	‘[pvt|priv]+*[cert|key|rsa|secret|ssl]+’
$isUser()$	‘user’
$usesWeakAlgo()$	‘md5’, ‘sha1’

We made the smell identification task available to the raters using a website [3]. In each task, a rater determines which of the security smells identified in Section 8.2 occur in a script. We used graduate students from a graduate course conducted in 2019 as raters to construct the oracle dataset. We recruited these raters from a graduate-level course conducted in the university. Of the 60 students in the class, 32 students agreed to participate. We assigned 96 Ansible and 76 Chef scripts to the 32 students to ensure each script is reviewed by at least two students, where each student does not have to rate more than 15 scripts. We used balanced block design to assign 96 Ansible and 76 Chef scripts from our collection of 1,101 Ansible and 855 Chef scripts.

For Puppet, we raters from a graduate-level course conducted in 2018. We obtained institutional review board (IRB) approval for the student participation. Of the 58 students in the class, 28 students agreed to participate. We assigned 140 scripts to the 28 students to ensure each script is reviewed by at least two students, where each student does not have to rate more than 10 scripts. We used balanced block design to assign 140 scripts from our collection of 1,726 scripts. We observe agreements on the rating for 79 of 140 scripts (56.4%), with a Cohen's Kappa of 0.3. According to Landis and Koch's interpretation [LK77], the reported agreement is ‘fair’. In the case of disagreements between raters for 61 scripts, the dissertation author resolved the disagreements.

Upon completion of the oracle dataset, we evaluate the accuracy of SLIC using precision and recall for the oracle dataset. Precision refers to the fraction of correctly identified smells among the total identified security smells, as determined by SLIC. Recall refers to the fraction of correctly identified smells that have been retrieved by SLIC over the total amount of security smells.

Performance of SLIC for Oracle Dataset: We report the detection accuracy of SLIC with respect to precision and recall For Ansible in Table 8.8. As shown in the ‘No smell’ row, we identify 77 Ansible scripts with no security smells. For example, in the oracle dataset, we identify one occurrence of

[3]http://13.59.115.46/website/start.php

‘Empty password’. The count of occurrences for each security smell along with SLIC’s precision and recall for the Ansible and Chef oracle dataset are provided in Table 8.8 and 8.9.

For Puppet we report the detection accuracy of SLIC with respect to precision and recall in Table 8.7. As shown in the ‘No smell’ row, we identify 113 scripts with no security smells. The rest of the 27 scripts contained at least one occurrence of the seven smells. The count of occurrences for each security smell along with SLIC’s precision and recall for the oracle dataset are provided in Table 8.7. For example, in the oracle dataset, we identify one occurrence of ‘Admin by default’ smell. The precision and recall of SLIC for one occurrence of admin by default is respectively, 1.0 and 1.0. SLIC generates zero false positives and one false negative for ‘Hard-Coded secret’. For the oracle dataset average precision and recall of SLIC is 0.99.

Table 8.7 SLIC’s Accuracy for the Puppet Oracle Dataset

Smell Name	Occurr.	Precision	Recall
Admin by default	1	1.00	1.00
Empty password	2	1.00	1.00
Hard-coded secret	24	1.00	0.96
Invalid IP address binding	4	1.00	1.00
Suspicious comment	17	1.00	1.00
Use of HTTP without TLS	9	1.00	1.00
Use of weak crypto. algo.	1	1.00	1.00
No smell	113	0.99	1.00
Average		0.99	0.99

Table 8.8 SLIC’s Accuracy for the Ansible Oracle Dataset

Smell Name	Occurr.	Precision	Recall
Empty password	1	1.0	1.0
Hard-coded secret	1	1.0	1.0
Invalid IP address binding	2	1.0	1.0
Suspicious comment	4	1.0	1.0
Use of HTTP without TLS	14	1.0	1.0
No Integrity Check	2	1.0	1.0
No smell	77	1.0	1.0
Average		1.0	1.0

8.2.4 RQ2: How frequently do security anti-patterns occur?

RQ2 focuses on characterizing how frequently security smells are present. First, we apply SLIC to determine the security smell occurrences for each script. Second, we calculate two metrics described

Table 8.9 SLIC's Accuracy for the Chef Oracle Dataset

Smell Name	Occurr.	Precision	Recall
Admin by default	2	1.0	1.0
Hard-coded secret	25	0.8	1.0
Invalid IP address binding	1	1.0	1.0
Suspicious comment	10	1.0	1.0
Use of HTTP without TLS	27	1.0	1.0
Use of weak crypto. algo.	2	1.0	1.0
No smell	61	1.0	0.9
Average		0.9	0.9

below:

- ***Smell Density***: Similar to prior research that have used defect density [Kel92] and vulnerability density [AM05], we use smell density to measure the frequency of a security smell x, for every 1000 lines of code (LOC). We measure smell density using Equation 8.1.

$$\text{Smell Density } (x) = \frac{\text{Total occurrences of } x}{\text{Total line count for all scripts/1000}} \tag{8.1}$$

- ***Proportion of Scripts (Script%)***: Similar to prior work in defect analysis [RW18], we use the metric 'Proportion of Scripts' to quantify how many scripts have at least one security smell. This metric refers to the percentage of scripts that contain at least one occurrence of smell x.

The two metrics characterize the frequency of security smells differently. The smell density metric is more granular, and focuses on the content of a script as measured by how many smells occur for every 1000 LOC. The proportion of scripts metric is less granular and focuses on the existence of at least one of the seven security smells for all scripts.

We conduct our empirical study with four datasets of Puppet scripts. Three datasets are constructed using repositories collected from three organizations: Mozilla, Openstack, and Wikimedia. The fourth dataset is constructed from repositories hosted on GitHub. To assess the prevalence of the identified smells and increase generalizability of our findings, we include repositories from Github, as companies tend to host their popular OSS projects on GitHub [Kri18] [Agr18].

As advocated by prior research [Mun17], OSS repositories need to be curated. We apply the following criteria to curate the collected repositories:

- **Criteria-1**: At least 11% of the files belonging to the repository must be IaC scripts. Jiang and Adams [JA15] reported for OSS repositories, which are used in production, IaC scripts co-exist with other types of files, such as Makefiles. They observed a median of 11% of the files to be IaC

Table 8.10 Puppet Repositories Satisfying Criteria

	GH	MOZ	OST	WIK
Initial Repo Count	3,405,303	1,594	1,253	1,638
Criteria-1 (11% IaC Scripts)	6,088	2	67	11
Criteria-2 (Not a Clone)	4,040	2	61	11
Criteria-3 (Commits/Month ≥ 2)	2,711	2	61	11
Criteria-4 (Contributors $\geq$ 10)	219	2	61	11
Final Repo Count	219	2	61	11

scripts. By using a cutoff of 11% we assume to collect repositories that contain sufficient amount of IaC scripts for analysis.

- **Criteria-2**: The repository is not a clone.
- **Criteria-3**: The repository must have at least two commits per month. Munaiah et al. [Mun17] used the threshold of at least two commits per month to determine which repositories have enough software development activity. We use this threshold to filter repositories with little activity.
- **Criteria-4**: The repository has at least 10 contributors. Our assumption is that the criteria of at least 10 contributors may help us to filter out irrelevant repositories. Previously, researchers have used the cutoff of at least nine contributors [Kri18] [Agr18] [Rah18c].

As shown in Table 8.10, we answer RQ2 using 15,232 scripts collected from 219, 2, 61, and 11 repositories, respectively, from GitHub, Mozilla, Openstack, and Wikimedia. We clone the master branches of the 293 repositories. Summary attributes of the collected repositories are available in Table 8.11.

Table 8.11 Summary Attributes of the Puppet Datasets

Attribute	**GH**	**MOZ**	**OST**	**WIK**
Repository Type	Git	Mercurial	Git	Git
Repository Count	219	2	61	11
Total File Count	72,817	9,244	12,681	9,913
Total Puppet Scripts	8,010	1,613	2,764	2,845
Tot. LOC (Puppet Scripts)	424,184	66,367	214,541	135,137

As shown in Table 8.12, we answer RQ2 using 14,253 Ansible and 36,070 Chef scripts respectively, from 365 and 448 repositories. We clone the master branches of the 365 Ansible and 448 Chef repositories. Summary attributes of the collected repositories are available in Table 8.13.

Table 8.12 Ansible and Chef Repositories Satisfying Criteria (Sect. 8.2.4)

	Ansible		Chef	
	GH	OST	GH	OST
Initial Repo Count	3,405,303	1,253	3,405,303	1,253
Criteria-1 (11% IaC Scripts)	13,768	16	5,472	15
Criteria-2 (Not a Clone)	10,017	16	3,567	11
Criteria-3 (Commits/Month ≥ 2)	10,016	16	3,565	11
Criteria-4 (Devs ≥ 10)	349	16	438	10
Final Repo Count	349	16	438	10

Table 8.13 Summary Attributes of the Datasets

	Ansible		**Chef**	
Attribute	**GH**	**OST**	**GH**	**OST**
Repository Count	349	16	438	11
Total File Count	498,752	4,487	126,958	2,742
Total Script Count	13,152	1,101	35,132	938
Tot. LOC (IaC Scripts)	602,982	52,239	1,981,203	63,339

8.2.5 RQ3: How do practitioners perceive the identified security anti-patterns?

We gather feedback using bug reports on how practitioners perceive the identified security smells. From the feedback we can assess if the identified security smells actually have an impact on how practitioners develop IaC scripts. We apply the following procedure:

First, we randomly select 1,000 occurrences of security smells from the four datasets. **Second**, we post a bug report for each occurrence, describing the following items: smell name, brief description, related CWE, and the script and line number where the smell occurred. We explicitly ask if contributors of the repository agrees to fix the smell instances.

8.3 Results

In this section, we provide answers to the four research questions:

8.3.1 RQ1: What security smells occur in infrastructure as code scripts?

Using our methodology we identify seven security smells for Puppet, eight security smells for Chef, and six security smells for Ansible. Each of the security smells along with the applicable programming language is presented in this section. The names of the smells are presented alphabetically. Examples of each security smell are presented in Figures 8.5, 8.6, and 8.7 respectively for Puppet, Ansible, and Chef.

Admin by default[Chef, Puppet]: This smell is the recurring pattern of specifying default users as administrative users. The smell can violate the 'principle of least privilege' property [Nat14], which recommends practitioners to design and implement a system in a manner so that by default the least amount of access necessary is provided to any entity. In Figure 8.5, two of the default parameters are '$power_username', and '$power_password'. If no values are passed to this script, then the default user will be 'admin', and can have full access. The smell is related with 'CWE-250: Execution with Unnecessary Privileges' [MIT18].

Empty password[Ansible, Puppet]: This smell is the recurring pattern of using a string of length zero for a password. An empty password is indicative of a weak password. An empty password does not always lead to a security breach, but makes it easier to guess the password. The smell is similar to the weakness 'CWE-258: Empty Password in Configuration File' [MIT18]. An empty password is different from using no passwords. In SSH key-based authentication, instead of passwords, public and private keys can be used [YL06]. Our definition of empty password does not include usage of no passwords and focuses on attributes/variables that are related to passwords and assigned an empty string. Empty passwords are not included in hard-coded secrets because for a hard-coded secret, a configuration value must be a string of length one or more.

Hard-coded secret[Ansible, Chef, Puppet]: This smell is the recurring pattern of revealing sensitive information such as user name and passwords as configurations in IaC scripts. IaC scripts provide the opportunity to specify configurations for the entire system, such as configuring user name and password, setting up SSH keys for users, specifying authentications files (creating key-pair files for Amazon Web Services). However, in the process programmers can hard-code these pieces of information into scripts. In Figure 8.5, we provide six examples of hard-coded secrets. We consider three types of hard-coded secrets: hard-coded passwords, hard-coded user names, and hard-coded private cryptography keys. Relevant weaknesses to the smell are 'CWE-798: Use of Hard-coded Credentials' and 'CWE-259: Use of Hard-coded Password' [MIT18]. For source code, practitioners acknowledge the existence of hard-coded secrets and advocate for tools such as CredScan [4] to scan source code.

We acknowledge that practitioners may intentionally leave hard-coded secrets such as user names and SSH keys in scripts, which may not be enough to cause a security breach. Hence this practice is security smell, but not a vulnerability.

Invalid IP address binding[Ansible, Chef, Puppet]: This smell is the recurring pattern of assigning the address 0.0.0.0 for a database server or a cloud service/instance. Binding to the address 0.0.0.0 may cause security concerns as this address can allow connections from every possible network [Mut99]. Such binding can cause security problems as the server, service, or instance will be exposed to all IP addresses for connection. For example, practitioners have reported how binding to

[4]https://blogs.msdn.microsoft.com/visualstudio/2017/11/17/managing-secrets-securely-in-the-cloud/

0.0.0.0 facilitated security problems for MySQL [5] (database server), Memcached [6] (cloud-based cache service) and Kibana [7] (cloud-based visualization service). We acknowledge that an organization can opt to bind a database server or cloud instance to 0.0.0.0, but this case may not be desirable overall. This smell is related to improper access control as stated in the weakness 'CWE-284: Improper Access Control' [MIT18].

Missing Default in Case Statement[Chef]: This smell is the recurring pattern of not handling all input combinations when implementing a case conditional logic. Because of this coding pattern, an attacker can guess a value, which is not handled by the case conditional statements and trigger an error. Such error can provide the attacker unauthorized information of the system in terms of stack traces or system error. This smell is related to 'CWE-478: Missing Default Case in Switch Statement' [MIT18].

No integrity check[Ansible, Chef]: This smell is the recurring pattern of not checking repository content that is being downloaded using checksums and gpg signatures. By not checking for integrity, a developer assumes the downloaded content is secure and has not been corrupted by a potential attacker. Checking for integrity provides an additional layer of security to ensure that the downloaded content is intact, and the downloaded link has not been compromised by an attacker, possibly inserting a virus payload. This smell is related to 'CWE-353: Missing Support for Integrity Check' [MIT18].

Suspicious comment[Ansible, Chef, Puppet]: This smell is the recurring pattern of putting information in comments about the presence of defects, missing functionality, or weakness of the system. The smell is related to 'CWE-546: Suspicious Comment' [MIT18]. Examples of such comments include putting keywords such as 'TODO', 'FIXME', and 'HACK' in comments, along with putting bug information in comments. Keywords such as 'TODO' and 'FIXME' in comments are used to specify an edge case or a problem [Sto08]. However, these keywords make a comment 'suspicious' i.e., indicating missing functionality about the system.

Use of HTTP without TLS[Ansible, Chef, Puppet]: This smell is the recurring pattern of using HTTP without the Transport Layer Security (TLS). Such use makes the communication between two entities less secure, as without TLS, use of HTTP is susceptible to man-in-the-middle attacks [Res00]. For example, as shown in Figure 8.5, the authentication protocol is set to 'http' for the branch that satisfies the condition 'RedHat'. Such usage of HTTP can be problematic, as the 'admin-user' will be connecting over a HTTP-based protocol. An attacker can eavesdrop on the communication channel and may guess the password of user 'admin-user'. This security smell is related to 'CWE-319: Cleartext Transmission of Sensitive Information' [MIT18]. The motivation for using HTTPS is to protect the privacy and integrity of the exchanged data. Information sent over HTTP may be

[5] https://serversforhackers.com/c/mysql-network-security
[6] https://news.ycombinator.com/item?id=16493480
[7] https://www.elastic.co/guide/en/kibana/5.0/breaking-changes-5.0.html

encrypted, and in such case 'Use of HTTP without TLS' may not lead to a security attack.

Use of weak cryptography algorithms[Ansible, Puppet]: This smell is the recurring pattern of using weak cryptography algorithms, such as MD4 and SHA-1 for encryption purposes. MD5 suffers from security problems, as demonstrated by the Flame malware in 2012 [CC12]. MD5 is susceptible to collision attacks [BB94] and modular differential attacks [WY05]. In Figure 8.5, we observe a password is being set using the 'ht_md5' method provided by the 'htpasswd' Puppet module [8]. Similar to MD5, SHA1 is also susceptible to collision attacks [9]. This smell is related to 'CWE-327: Use of a Broken or Risky Cryptographic Algorithm' and 'CWE-326: Inadequate Encryption Strength' [MIT18]. When weak algorithms such as MD5 are used for hashing that may not lead to a breach, but using MD5 for password setup may.

8.3.2 RQ2: How frequently do security anti-patterns occur?

We observe our identified security smells to exist across all datasets. For GitHub, Mozilla, Openstack, and Wikimedia respectively, 29.3%, 17.9%, 32.9%, and 26.7% of all scripts include at least one occurrence of our identified smells. Hard-coded secret is the most prevalent security smell with respect to occurrences and smell density. Altogether, we identify 16,952 occurrences of hard-coded secrets, of which 68.3%, 23.9%, and 7.8% are respectively, hard-coded keys, user names, and passwords. A complete breakdown of findings related to RQ2 is presented in Table 8.15 for our four datasets GitHub ('GH') Mozilla ('MOZ'), Openstack ('OST'), and Wikimedia ('WIK').

For Ansible, in our GitHub and Openstack datasets we observe respectively 25.3% and 29.6% of the total scripts to contain at least one of the six identified security smells. For Chef, in our GitHub and Openstack datasets we observe respectively 20.5% and 30.4% of the total scripts to contain at least one of the eight identified security smells. Hard-coded secret is the most prevalent security smell with respect to occurrences, smell density, and proportion of scripts contain hard-coded secrets. For Ansible we identify 15,131 occurrences of hard-coded secrets, of which 55.9%, 37.0%, and 7.1% are respectively, hard-coded keys, user names, and passwords. For Chef we identify 15,363 occurrences of hard-coded secrets, of which 47.0%, 8.9%, and 44.1% are respectively, hard-coded keys, user names, and passwords. A complete breakdown of findings related to RQ2 is presented in Table 8.15 for our two datasets GitHub ('GH') and Openstack ('OST'), for both Ansible and Chef.

Occurrences: The occurrences of the seven security smells are presented in the 'Occurrences' column of Table 8.15. The 'Combined' row presents the total smell occurrences. For Github, Mozilla, Openstack, and Wikimedia we respectively observe 13221, 1141, 4507, and 2332 occurrences of security smells.

The occurrences of the security smells are presented in the 'Occurrences' column of Table 8.15.

[8]https://forge.puppet.com/leinaddm/htpasswd
[9]https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html

The ‘Combined’ row presents the total smell occurrences. In the case of Ansible scripts, we observe 18,353 occurrences of security smells, and for Chef, we observe 28,404 occurrences of security smells.

Table 8.14 Smell Occurrences, Smell Density, and Proportion of Scripts for the Four Datasets of Puppet Scripts

	Occurrences				Smell Density (per KLOC)				Proportion of Scripts (Script%)			
Smell Name	**GH**	**MOZ**	**OST**	**WIK**	**GH**	**MOZ**	**OST**	**WIK**	**GH**	**MOZ**	**OST**	**WIK**
Admin by default	52	4	35	6	0.1	0.06	0.1	0.04	0.6	0.2	1.1	0.2
Empty password	136	18	21	36	0.3	0.2	0.1	0.2	1.4	0.4	0.5	0.3
Hard-coded secret	10,892	792	3,552	1,716	25.6	11.9	16.5	12.7	21.9	9.9	24.8	17.0
Invalid IP address binding	188	20	114	41	0.4	0.3	0.5	0.3	1.7	0.7	2.9	1.4
Suspicious comment	758	202	305	343	1.7	3.0	1.4	2.5	5.9	8.5	7.2	9.1
Use of HTTP without TLS	1,018	57	460	164	2.4	0.8	2.1	1.2	6.3	1.6	8.5	3.7
Use of weak crypto algo.	177	48	20	26	0.4	0.7	0.1	0.2	0.9	1.1	0.5	0.4
Combined	13,221	1,141	4,507	2,332	31.1	17.2	21.0	17.2	29.3	17.9	32.9	26.7

Table 8.15 Smell Occurrences, Smell Density, and Proportion of Scripts for Ansible and Chef scripts

	Occurrences				Smell Density (per KLOC)				Proportion of Scripts (Script%)			
	Ansible		Chef		Ansible		Chef		Ansible		Chef	
Smell Name	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**
Admin by default	N/A	N/A	301	61	N/A	N/A	0.1	0.9	N/A	N/A	0.3	2.1
Empty password	298	3	N/A	N/A	0.49	0.06	N/A	N/A	1.1	0.2	N/A	N/A
Hard-coded secret	14,409	722	14,160	1,203	23.9	13.8	7.1	19.0	19.2	22.4	6.8	15.9
Invalid IP address binding	129	7	591	19	0.2	0.1	0.3	0.3	0.5	0.4	1.1	1.0
Missing default in switch	N/A	N/A	953	68	N/A	N/A	0.5	1.0	N/A	N/A	2.5	6.5
No integrity check	194	14	2,249	132	0.3	0.2	1.1	2.1	1.1	1.0	3.6	3.8
Suspicious comment	1,421	138	3,029	161	2.3	2.6	1.5	2.5	6.3	8.0	6.6	9.3
Use of HTTP without TLS	934	84	4,898	326	1.5	1.6	2.4	5.1	3.7	3.0	4.9	6.9
Use of weak crypto algo.	N/A	N/A	94	2	N/A	N/A	0.05	0.03	N/A	N/A	0.2	0.1
Combined	17,385	968	26,407	1,997	28.8	18.5	13.3	31.5	25.3	29.6	20.5	30.4

Smell Density: In the ‘Smell Density (per KLOC)’ column of Table 8.15 we report the smell density. The ‘Combined’ row presents the smell density for each dataset when all seven security smell occurrences are considered. For all Ansible, Chef, and Puppet datasets, we observe the dominant security smell is ‘Hard-coded secret’. The least dominant smell is ‘Admin by default’.

Security smells occur more frequently for Puppet scripts hosted in GitHub. For hard-coded secrets, smell density is 1.5 ∼ 2.1 times higher for scripts hosted in GitHub than the other three datasets.

Proportion of Scripts (Script%): In the ‘Proportion of Scripts (Script%)’ column of Table 8.15, we report the proportion of scripts (Script %) values for each of the four datasets. The ‘Combined’ row represents the proportion of scripts in which at least one of the identified smells appear. As shown in the ‘Combined’ row, for Ansible percentage of scripts that have at least one of six smells is respectively, 25.3% and 29.6% for GitHub and Openstack. For Chef, percentage of scripts that have at least one of eight smells is respectively, 25.3% and 29.6% for GitHub and Openstack.

In the ‘Proportion of Scripts (Script%)’ column of Table 8.15, we report the proportion of scripts

(Script %) values for each of the four Puppet datasets. The 'Combined' row represents the proportion of scripts in which at least one of the seven smells appear. As shown in the 'Combined' row, percentage of scripts that have at least one of seven smells is respectively, 29.3%, 17.9%, 32.9%, and 26.7% for GitHub, Mozilla, Openstack, and Wikimedia.

8.3.3 RQ3: How do practitioners perceive the identified security anti-patterns?

From 7 and 19 repositories respectively, we obtain 29 and 54 responses for Ansible and Chef scripts. In the case of Ansible scripts we observe an agreement of 82.7% for 29 smell occurrences. For Chef scripts, we observe an agreement of 55.5% for 54 smell occurrences. The percentage of smells to which practitioners agreed to be fixed for Ansible and Chef is respectively, presented in Figure 8.8 and 8.9. In the y-axis each smell name is followed by the occurrence count. For example, according to Figure 8.8, for 4 occurrences of 'Use of HTTP without TLS' (HTTP.USG) , we observe 100% agreement for Ansible scripts.

From 21 repositories we obtain 104 responses for the submitted 500 bug reports in terms of acknowledgement or changing the source code. We observe an agreement of 63.4% for 104 smell occurrences. The percentage of smells to which practitioners agreed to be fixed is presented in Figure 8.10. In the y-axis each smell name is followed by the occurrence count. For example, according to Figure **??**, for 9 occurrences of 'Use of weak cryptography algorithms'(WEAK_CRYP), we observe 77.7% agreement. We observe 75.0% or more agreement for one smell: 'Use of weak cryptography algorithms'.

Reasons for Practitioner Agreements: In their response, practitioners provided reasoning on why these smells appeared. For one occurrence of 'HTTP without TLS' in a Chef script, one practitioner suggested availability of a HTTPS endpoint saying: "*In this case, I think it was just me being a bit sloppy: the HTTPS endpoint is available so I should have used that to download RStudio packages from the start*". For an occurrence of hard-coded secret in an Ansible script one practitioner agreed stating possible solutions: "*I agree that it [hard-coded secret] could be in an Ansible vault or something dedicated to secret storage.*". Upon acceptance of the smell occurrences, practitioners also suggested how these smells can be mitigated. For example, for an occurrence of 'Invalid IP address binding' in a Puppet script, one practitioner stated:"*I would accept a pull request to do a default of 127.0.0.1*".

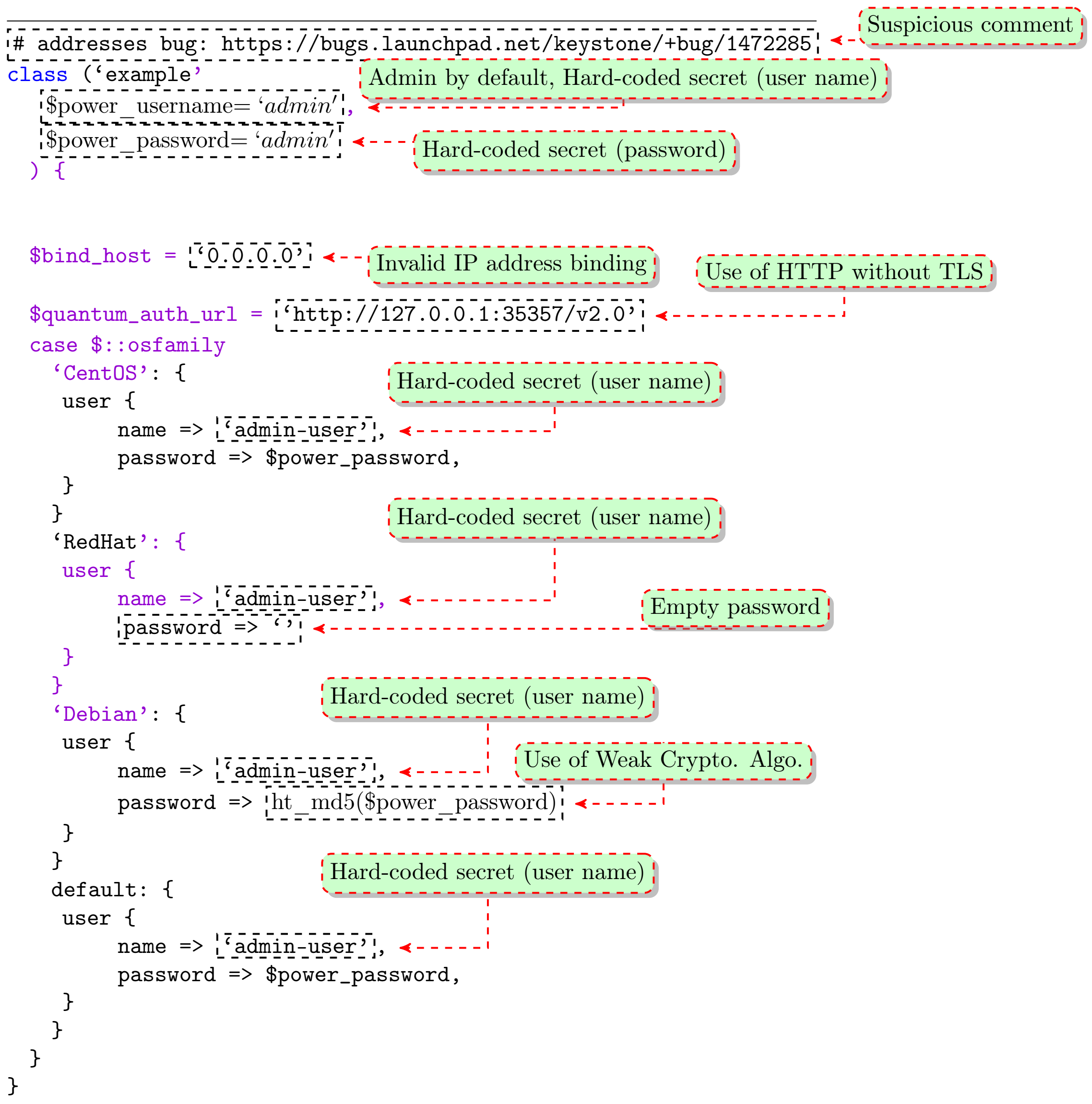
Reasons for Practitioner Disagreements: We observe practitioners to have strong perceptions on whether a security smell is actually security-related. One practitioner disagreed with an occurrence of 'suspicious comment' stating "*Please do not file vague issue like this going forward*". Context is also important to practitioners. For example, a hard-coded password may not have security implications for practitioners if the hard-coded password is used for testing purposes. One practitioner disagreed stating "*the code in question is an integration test. The username and password is not used anywhere else so this should be no issue.*". These anecdotal evidence suggests that while developing

IaC scripts practitioners may only be considering their own development context, and not realizing how another practitioner may perceive use of these security smells as an acceptable practice. For one occurrence of 'HTTP Without TLS' in a Puppet script one practitioner disagreed stating "*It's using http on localhost, what's the risk?*".

The above-mentioned statements from disagreeing practitioners also suggest lack of awareness: the users who use the training module of interest may consider use of hard-coded passwords as an acceptable practice, potentially propagating the practice of hard-coded secrets. Both local and remote sites that use HTTP can be insecure, as considered by practitioners from Google [10] [11]. Possible explanations for disagreements can also be attributed to perception of practitioners: smells in code have subjective interpretation [Hal14], and programmers do not uniformly agree with all smell occurrences [ML06; Pal14]. Furthermore, researchers [Aca17a] have observed programmers' bias to perceive their code snippets as secure, even if the code snippets are insecure.

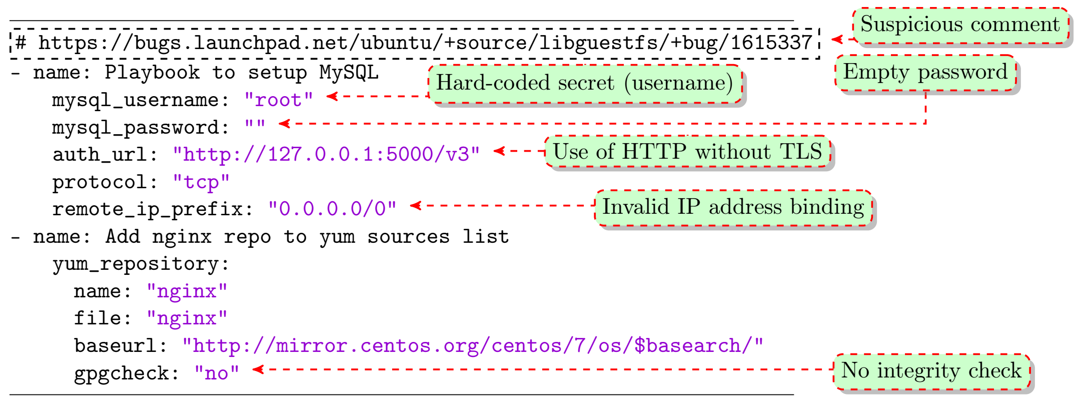
Summary: We identify nine security smells from our qualitative analysis of Ansible, Chef, and Puppet scripts. Using SLIC, our static analysis tool, we identify 18,353, 28,404, and 21,201 occurrences occurrences of security smells respectively, for Ansible, Chef, and Puppet. We observe 1,077, 6,788, and 1,326 hard-coded passwords to be included in the identified security smells respectively, for Ansible, Chef, and Puppet. We observe an agreement rate of 82.7%, 74.0%, and 69.8% respectively, for the acknowledged 29, 27, and 212 bug reports for Ansible, Chef, and Puppet, which suggests the relevance of security smells for IaC scripts amongst practitioners.

[10]https://security.googleblog.com/2018/02/a-secure-web-is-here-to-stay.html
[11]https://developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https


Suspicious comment
Admin by default, Hard-coded secret (user name)
Hard-coded secret (password)
Invalid IP address binding
Use of HTTP without TLS
Hard-coded secret (user name)
Hard-coded secret (user name)
Empty password
Hard-coded secret (user name)
Use of Weak Crypto. Algo.
Hard-coded secret (user name)

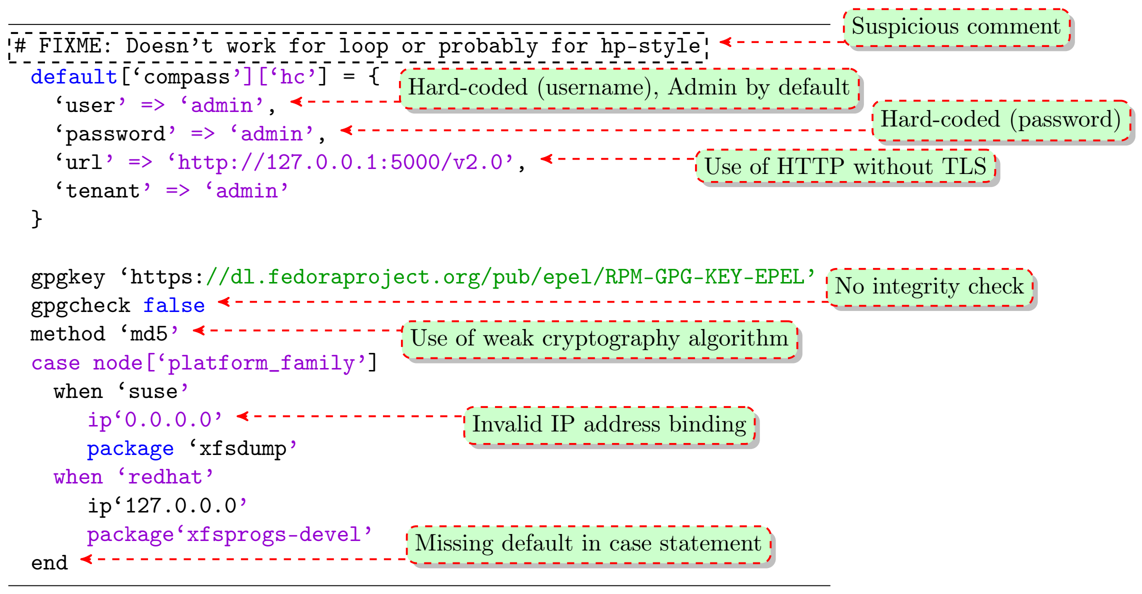
```
# addresses bug: https://bugs.launchpad.net/keystone/+bug/1472285
class (‘example’
   $power_username= ‘admin’,
   $power_password= ‘admin’
  ) {


  $bind_host = ‘0.0.0.0’

  $quantum_auth_url = ‘http://127.0.0.1:35357/v2.0’
  case $::osfamily
    ‘CentOS’: {
     user {
          name => ‘admin-user’,
          password => $power_password,
     }
    }
    ‘RedHat’: {
     user {
          name => ‘admin-user’,
          password => ‘’
     }
    }
    ‘Debian’: {
     user {
          name => ‘admin-user’,
          password => ht_md5($power_password)
     }
    }
    default: {
     user {
          name => ‘admin-user’,
          password => $power_password,
     }
    }
  }
}
```

Figure 8.5 An annotated script with all seven security smells for Puppet. The name of each security smell is highlighted on the right.



```
# https://bugs.launchpad.net/ubuntu/+source/libguestfs/+bug/1615337
- name: Playbook to setup MySQL
    mysql_username: "root"
    mysql_password: ""
    auth_url: "http://127.0.0.1:5000/v3"
    protocol: "tcp"
    remote_ip_prefix: "0.0.0.0/0"
- name: Add nginx repo to yum sources list
    yum_repository:
      name: "nginx"
      file: "nginx"
      baseurl: "http://mirror.centos.org/centos/7/os/$basearch/"
      gpgcheck: "no"
```

Figure 8.6 An annotated Ansible script with six security smells. The name of each security smell is highlighted on the right.



```
# FIXME: Doesn't work for loop or probably for hp-style
  default['compass']['hc'] = {
    'user' => 'admin',
    'password' => 'admin',
    'url' => 'http://127.0.0.1:5000/v2.0',
    'tenant' => 'admin'
  }

  gpgkey 'https://dl.fedoraproject.org/pub/epel/RPM-GPG-KEY-EPEL'
  gpgcheck false
  method 'md5'
  case node['platform_family']
    when 'suse'
      ip'0.0.0.0'
      package 'xfsdump'
    when 'redhat'
      ip'127.0.0.0'
      package'xfsprogs-devel'
  end
```

Suspicious comment
Hard-coded (username), Admin by default
Hard-coded (password)
Use of HTTP without TLS
No integrity check
Use of weak cryptography algorithm
Invalid IP address binding
Missing default in case statement

Figure 8.7 An annotated Chef script with eight security smells. The name of each security smell is highlighted on the right.

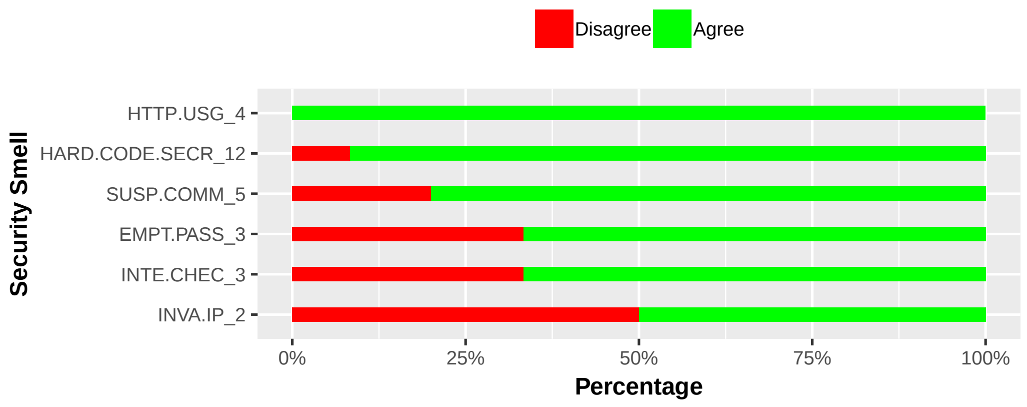


Figure 8.8 Feedback for 29 smell occurrences for Ansible. Practitioners agreed with 82.7% of the selected smell occurrences.

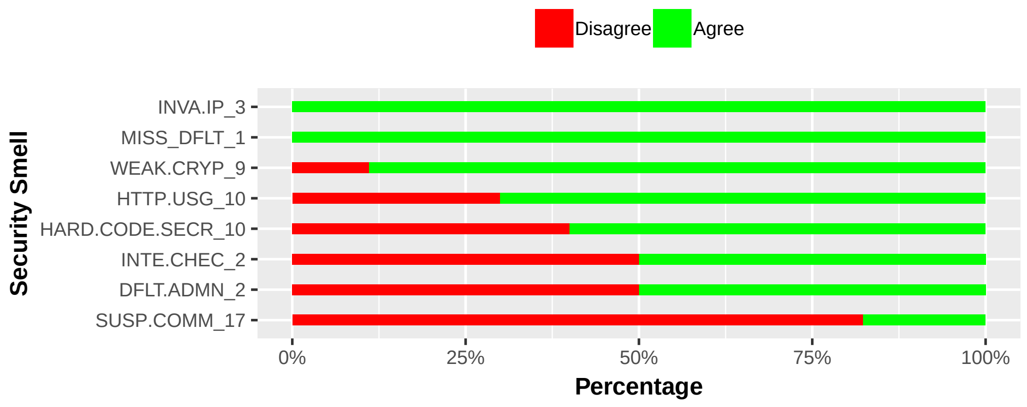


Figure 8.9 Feedback for the 54 smell occurrences for Chef. Practitioners agreed with 55.5% of the selected smell occurrences.

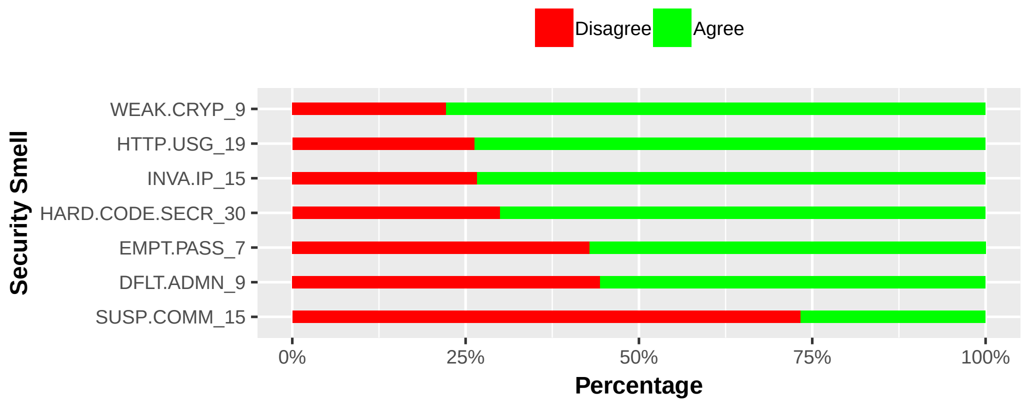


Figure 8.10 Feedback for the 212 smell occurrences. Practitioners agreed with 69.8% of the selected smell occurrences.

CHAPTER

9 DISCUSSION

We discuss our empirical findings in this chapter.

9.1 Defect Categories in Infrastructure as Code Scripts

One finding from Chapter 4 is the prevalence of assignment-related defects in IaC scripts. Software teams can use this finding to improve their process in two possible ways: first, they can use the practice of code review for developing IaC scripts. Code reviews can be conducted using automated tools and/or team members' manual reviews. For example, through code reviews software teams can pinpoint the correct value of configurations at development stage. Automated code review tools such as linters can also help in detecting and fixing syntax issues of IaC scripts at the development stage. Typically, IaC scripts are used by organizations that have implemented CD, and for these organizations, Kim et al. [Kim16] recommends manual peer review methods such as pair programming to improve code quality.

Second, software teams might benefit from unit testing of IaC scripts to reduce defects related to configuration assignment. We have observed from Chapter 4 that IaC-related defects are mostly related to the assignment category which includes improper assignment of configuration values and syntax errors. Practitioners can test if correct configuration values are assigned by writing unit tests for components of IaC scripts. In this manner, instead of catching defects at run-time that might lead to real-world consequences e.g. the problem reported by Wikimedia Commons [1], software teams might be able to catch defects in IaC scripts at the development stages with the help of testing.

9.2 Operations that Characterize Defective Scripts

We discuss our findings of Chapter 5 with possible implications:

- **Prioritizing Verification and Validation Efforts**: One operation that characterizes defective IaC scripts is performing file-related operations. Erroneous file mode and file path can make filesystem operations more susceptible to defects. Filesystem operations are performed in 15.1%, 21.7%, 14.5%, and 23.4% of the scripts respectively for Mirantis, Mozilla, Openstack, and Wikimedia. To perform file operations practitioners have to provide configuration values such as file location and permissions. While assigning these values practitioners might be inadvertently making mistakes and introducing defects to IaC scripts. Software teams can take our findings into account, and prioritize their verification and validation efforts accordingly. They can write tests dedicated for scripts that are used to perform filesystem operations such as setting file locations and permissions. They can also benefit from extra inspection efforts to check if proper configuration values are being assigned in these particular scripts.

  Cito et al. [Cit15b] interviewed practitioners and observed that in cloud-based application development, use of IaC tools such as Puppet, is fundamental to automated provisioning of development and deployment infrastructure. Findings from our research provides further evidence to their observations. We also observe that infrastructure provisioning can be a source of defects for IaC scripts. Infrastructure provisioning using IaC scripts involves executing a sequence of complex steps, for example installation of third-party packages, ensuring scalability, and handling the sensitive information of systems [MJ11] [FP16] [Tur07]. While implementing these steps, practitioners might be introducing defects inadvertently. Infrastructure provisioning appears respectively for 6.9%, 18.9%, and 17.9% of Mozilla, Openstack, and Wikimedia scripts. Similar recommendations apply for managing user accounts as well. Similar to filesystem operations, IaC tools provide the options to setup and manage users [Lab17], and practitioners have to provide the proper configuration values in the required format. Our research indicates the practice of setting up user accounts is susceptible to defects, and IaC scripts that are used for user account management should be prioritized for more verification and validation. Compared to filesystem operations, scripts used for managing users is smaller: 2.5%, 1.1%, and 1.6% respectively, for Mozilla, Openstack, and Wikimedia.

- **Tools**: Our results show that text features can be a strategy to build defect prediction models for IaC scripts. Building defect prediction models for IaC scripts also provide the opportunity

[1]https://wikitech.wikimedia.org/wiki/Incident documentation/20170118-Labs

of creating new tools and services for IaC scripts. For software production code, such as C++ and Java code, tools and services exist that predict which source code files can be defect-prone. Toolsmiths can apply text mining on IaC scripts to build defect prediction models for IaC scripts.

9.3 Source Code Properties of Defective Scripts

We discuss our findings from Chapter 6 with possible implications as following:

- **Prioritization of Inspection Efforts**: Our findings have implications on how practitioners can prioritize inspection efforts for IaC scripts. The identified 12 source code properties can be helpful in early prediction of defective scripts. 'Hard-coded string' is correlated with making defective IaC scripts, and therefore, test cases can be designed by focusing on string-related values assigned in IaC scripts.

  Code inspection efforts can also be prioritized using our findings. According to our feature importance analysis, 'attribute' is correlated with defective IaC scripts. IaC scripts with relatively large amount of 'attributes' can get extra scrutiny. We observe other IaC-related source code properties that contribute to defective IaC scripts. Examples of such properties include: setting a file path ('file'), and executing external modules or scripts ('include'). Practitioners might benefit from code inspection using manual peer reviews for these particular properties as well.

  We also observe that IaC scripts can be as large; for example as large as 1,287 lines of code. To prioritize inspection efforts we advise practitioners to focus on the identified 12 source code properties. Large scripts further highlights the importance on identifying source code properties i.e. take a fine-grained approach.

- **Tools**: Prior research [Lew13] observed that defect prediction models can be helpful for programmers who write code in general purpose programming languages. Defect prediction of software artifacts is now offered as a cloud-service, as done by DevOps Insights [2]. For IaC scripts we observe the opportunity of creating a new set of tools and services that will help in defect mitigation. Toolsmiths can use our prediction models to build tools that pinpoint the defective IaC scripts that need to be fixed. Such tools can explicitly state which source code properties are more correlated with defects than others and need special attention when making changes. We recommend practitioners to explore source code-based properties along with process metrics, as process metrics might be better with respect to prediction performance.

[2]https://www.ibm.com/cloud/devops-insights

9.4 Security Anti-patterns

We suggest strategies on how the identified security anti-patterns can be mitigated along with other implications:

- **Admin by default**: We advise practitioners to create user accounts that have the minimum possible security privilege and use that account as default. Recommendations from Saltzer and Schroeder [SS75] may be helpful in this regard.
- **Empty password**: We advocate against storing empty passwords in IaC scripts. Instead, we suggest the use of strong passwords.
- **Hard-coded secret**: We suggest the following measures to mitigate hard-coded secrets:
  - use tools such as Vault [3] to store secrets
  - scan IaC scripts to search for hard-coded secrets using tools such as CredScan [4] and SLIC.
- **Invalid IP address binding**: To mitigate this anti-pattern, we advise programmers to allocate their IP addresses systematically based on which services and resources need to be provisioned. For example, incoming and outgoing connections for a database containing sensitive information can be restricted to a certain IP address and port.
- **Suspicious comment**: We acknowledge that in OSS development, programmers may be introducing suspicious comments to facilitate collaborative development and to provide clues on why the corresponding code changes are made [Sto08]. Based on our findings we advocate for creating explicit guidelines on what pieces of information to store in comments, and strictly follow those guidelines through code review. For example, if a programmer submits code changes where a comment contains any of the patterns mentioned for suspicious comments, the submitted code changes will not be accepted.
- **Use of HTTP without TLS**: We advocate companies to adopt the HTTP with TLS by leveraging resources provided by tool vendors, such as MySQL [5] and Apache [6]. We advocate for better documentation and tool support so that programmers do not abandon the process of setting up HTTP with TLS.

[3]https://www.vaultproject.io/
[4]https://secdevtools.azurewebsites.net/helpcredscan.html
[5]https://dev.mysql.com/doc/refman/5.7/en/encrypted-connections.html
[6]https://httpd.apache.org/docs/2.4/ssl/ssl_howto.html

- **Use of Weak cryptography algorithms**: We advise programmers to use cryptography algorithms recommended by the National Institute of Standards and Technology [Bar16] to mitigate this anti-pattern.

Guidelines: One possible strategy to mitigate security anti-patterns is to develop concrete guidelines on how to write IaC scripts in a secure manner. When constructing guidelines, the IaC community can take the findings of Acar et al. [Aca17b] into account, and include easy to understand, task-specific examples on how to write IaC scripts in a secure manner.

Perception: Overall feedback from the practitioners for the identified anti-patterns is nuanced. Feedback from practitioners are examples on how context and personal experience can constitute programmers' assessment of security impact. Our findings are consistent with prior work [Dev16], which reported practitioners to have strong opinions that are formed from personal experiences. We advocate for future research that will investigate on how programmers perceptions about security are created, and can be modified so that they adopt the best practices of secure coding.

Insecure Coding Practices: IaC scripts hosted on GitHub are susceptible to security anti-patterns. Compared to the other three datasets our GitHub dataset has higher occurrence of anti-patterns and anti-patterns density. Our findings suggest that IaC scripts found 'in the wild' do contain security anti-patterns that persist for a long time, for example for 98 months. Implication of this finding is that existence and persistence of these anti-patterns leave the possibility of another programmer using these scripts with security anti-patterns, potentially propagating use of insecure coding practices.

Tool Support: The constructed tool SLIC can be helpful for IT organizations who implement DevOps and wants to integrate security. Prior work [URW16] has reported that DevOps practitioners who integrate security into their organization perform automated code review. SLIC can be helpful in the automated code review phase, as SLIC analyzes IaC scripts automatically through static analysis.

9.5 Threats to validity

In this section we discuss the limitations of the conducted empirical studies.

Conclusion Validity: For dataset construction (Chapter 3), our approach is based on qualitative analysis, where raters categorized XCMs, and assigned defect categories. We acknowledge that the process is susceptible human judgment, and the raters' experience can bias the categories assigned. The accompanying human subjectivity can influence the distribution of the defect category for IaC scripts of interest. We mitigated this threat by assigning multiple raters for the same set of XCMs. Next, we used a resolver, who resolved the disagreements. Further, we cross-checked our categorization with practitioners who authored the XCMs, and observed 'substantial' to 'almost perfect' agreement.

In Chapter 8, the derived security smells and their association with CWEs are subject to the dissertation author's judgment. We account for this limitation by applying verification of CWE mapping with two raters excluding the dissertation author for Puppet, and by using another rater for Ansible and Chef. Also, the oracle dataset constructed by the raters is susceptible to subjectivity, as the raters' judgment influences appearance of a certain security smell.

Construct Validity: The constructed datasets presented in Chapter 3 are subject to bias of the raters who categorized the XCMs. We mitigate this bias by letting at least two raters review each XCM. For Openstack and Wikimedia around 50% of the XCMs were resolved by the dissertation author, which makes the final categorization biased.

Our process of using human raters to determine defect categories can be limiting, as the process is susceptible to mono-method bias, where subjective judgment of raters can influence the findings. We mitigated this threat by using multiple raters.

Also, for Mirantis and Wikimedia, we used graduate students who performed the categorization as part of their class work. Students who participated in the categorization process can be subject to evaluation apprehension, i.e. consciously or sub-consciously relating their performance with the grades they would achieve for the course. We mitigated this threat by clearly explaining to the students that their performance in the categorization process would not affect their grades.

The raters involved in the categorization process had professional experience in software engineering for at two years on average. Their experience in software engineering may make the raters curious about the expected outcomes of the categorization process, which may affect the distribution of the categorization process. Furthermore, the resolver also has professional experience in software engineering and IaC script development, which could influence the outcome of the defect category distribution.

In Chapter 5 we have used two text mining techniques, and we acknowledge that our use of two techniques is not comprehensive. We observe the opportunity to apply sophisticated text mining techniques, such as deep learning for text-based feature discovery. We also acknowledge, some defects such as incorrect file paths may not be captured using text features. We advocate for mining new sets of metrics such as code metrics, and process metrics, as text feature-based defect prediction can sometimes yield a median accuracy less than 60%.

In Chapters 5 and 6 we predict defects at the script level. Our analysis does not include which lines of an IaC script might be defective. In the future, we plan to create models that predict defects at the line level. Furthermore, in our paper, we have not investigated if the amount of text features have an impact of prediction performance, and will include this investigation in future.

External Validity: Our scripts for all five empirical studies are collected from the OSS domain and not from proprietary sources. Our findings are subject to external validity, as our findings may not be generalizable. For four empirical studies we construct our datasets using Puppet, which is a declarative language. Our findings may not generalize for IaC scripts that use an imperative form of language. We acknowledge that more datasets can help generalizing our findings. Also, the datasets do not include temporal information i.e. we do not account for presence or absence of defects across time. We plan to include more datasets in future that will also account for the temporal information for defects in IaC scripts.

Our findings are subject to external validity, as our findings may not be generalizable. We observe how security smells are subject to practitioner interpretation, and thus the relevance of security smells may vary from one practitioner to another. We construct our datasets using Puppet, which is a declarative language. Our findings may not generalize for IaC scripts that use an imperative form of language. Also, our scripts are collected from the OSS domain, and not from proprietary sources.

We acknowledge that our set of properties (Chapter 6), operations (Chapter 5), security anti-patterns (Chapter 8), and development anti-patterns (Chapter 7) is not comprehensive.

Internal Validity: As described in Chapter 3, we have used a combination of commit messages and issue report descriptions to determine if an IaC script is defective. We acknowledge that these messages might not have given the full context for the raters. Our set of metrics is also not comprehensive. We also acknowledge that in a defect-related commit message more than one defects can be fixed, but may not be expressed in the commit message and/or accompanied issue report description. Multiple raters may miss a defect-related commit, which may impact the distribution of the defect-related commits in our constructed datasets.

We have identified a set of code properties through qualitative analysis in Chapter 6. We derived these properties by applying qualitative analysis on defect-related commits of one dataset. We mitigated this limitation by applying empirical analysis on three more datasets, and quantify if the identified properties show correlation with defective scripts.

In Chapter 4 we have discussed how we used ODC to determine defect categories. We acknowledge that we have not used the trigger attribute of ODC, as our data sources do not include any information from which we can infer trigger attributes of ODC such as, 'design conformance', 'side effects', and 'concurrency'.

In we identify a set of security smells. We acknowledge that other security smells may exist. We mitigate this threat by manually analyzing 1,726 IaC scripts for security smells. In the future, we aim to investigate if more security smells exist. Also, the detection accuracy of SLIC depends on the constructed rules that we have provided in Chapter 8. We acknowledge that the constructed rules are heuristic-driven and susceptible to generating false positives and false negatives.

CHAPTER

10 FUTURE WORK AND CONCLUSION

We conclude the dissertation in this chapter with possible future research directions and conclusions.

10.1 Future Work

We describe possible future research directions based on our findings presented in this thesis.

Text Mining: We investigate two techniques to mine text features. Researchers can investigate if other techniques, such as topic modeling [Ble03] and word2vec [Mik13], can be applied to extract text mining features for defect prediction of IaC scripts. Future research can investigate how to improve the accuracy of text feature-based defect prediction models. Researchers can also investigate how text-based features compare with code metrics and process metrics.

Source Code Property: Our dissertation provides opportunity for further research in the area of defect prediction of IaC scripts. Sophisticated statistical techniques, such as, deep learning can be applied to discover more IaC-related source code properties. Researchers can investigate which automated program repair techniques is applicable for the identified source code properties. Researchers can also investigate how practitioners in real life perceive and use defect prediction models for IaC scripts.

Security Anti-pattern: Our findings show that not all IaC scripts include security smells. Researchers can build upon our findings to explore which characteristics correlate with IaC scripts with security smells. If certain characteristics correlate with scripts that have smells, then programmers can prioritize their inspection efforts for scripts that exhibit those characteristics. Researchers can also investigate if metric-based prediction techniques proposed in prior research [Rah17] can be used to identify IaC scripts that are more likely to include security smells, which can benefit from more scrutiny. Researchers can investigate how semantics and dynamic analysis of scripts can help in efficient smell detection. Researchers can also investigate what remediation strategies are needed to fix security smells.

Development Anti-pattern: Future researchers can investigate to what extent our findings generalize by conducting large-scale quantitative studies. Researchers can investigate the synergies between the development activities to see if combination of activities can improve quality of IaC scripts. Furthermore, researchers can use the seven development activity metrics to construct defect prediction models for IaC scripts. We hope our dissertation will facilitate further research in the domain of IaC script quality.

Dynamic Analysis: The conducted empirical studies in this thesis do not address the dynamic nature of IaC scripts. Researchers can investigate strategies on how correctness of configurations declared in IaC scripts can be verified at run-time and/or compile time. Researchers can investigate dynamic analysis techniques that identify best configuration settings and automatically generate IaC scripts. Furthermore, researchers can also investigate if our identified source code properties can be helpful in detecting configuration drift [Par17] and other system-level faults.

10.2 Conclusion

In continuous deployment, IT organizations rapidly deploy software and services to end-users using an automated deployment pipeline. IaC is a fundamental pillar to implement an automated deployment pipeline. Defective IaC scripts can hinder the reliability of the automated deployment pipeline. Characterizing source code properties of IaC scripts that correlate with defective IaC scripts can help identify signals to increase the quality and security of IaC scripts. *The goal of this thesis is to help practitioners in increasing quality of IaC scripts by identifying development and security anti-patterns in the development of infrastructure as code scripts.* We use IaC scripts collected from OSS repositories and conduct five empirical studies. In the first empirical study by applying the ODC technique, we observe defect category distribution for IaC to be different from non-IaC software systems. In the second empirical study, by applying text mining techniques and qualitative analysis we identify three properties that characterize defective scripts: filesystem operations, infrastructure provisioning, and managing user accounts. In the third empirical study, we apply qualitative analysis on IaC-related code changes to identify 12 source code properties that correlate with defective IaC scripts. We observe 10 of the 12 properties to show correlation with defective IaC scripts for all four datasets. The properties that show the strongest correlation are 'lines of code' and 'hard-coded string'. In contrast to our empirical analysis, we observe practitioners to agree most with the 'URL' property.

The identified operations and source code properties that relate with defective scripts may not be directly actionable. We mitigate this limitation by conducting two more empirical studies. In the fourth empirical study, we focus on identifying development activities that relate with defective scripts. We conduct a mixed-methods study with 2,138 IaC scripts, where we apply quantitative and qualitative analysis to determine development anti-patterns. We identify 13 development anti-patterns: 5 from our quantitative analysis and 8 from practitioner interviews. Our findings suggest defective scripts are related to traditional development activities, such as the number of developers working on a script, along with activities specific to continuous deployment, such as lack of feature flags usage. Our identified development anti-patterns suggest the importance of 'as code' activities such as adequate logging and adequate testing in IaC because these activities are related to defective IaC scripts.

In the fifth empirical study, we focus on security-specific anti-patterns. We identify security anti-patterns i.e. coding patterns that are indicative of security weaknesses. Security anti-patterns are recurring coding patterns in IaC scripts that are indicative of security weakness and can potentially lead to security breaches. By applying qualitative analysis on 1,726 scripts we identified seven security anti-patterns: admin by default; empty password; hard-coded secret; invalid IP address binding; suspicious comment; use of HTTP without TLS; and use of weak cryptography algorithms. We analyzed 15,232 IaC scripts to determine which security anti-patterns occur and used a tool called SLIC, to automatically identify security anti-patterns that occur in IaC scripts. We evaluated SLIC's accuracy by constructing an oracle dataset. We identified 21,201 occurrences of security anti-patterns that included 1,326 occurrences of hard-coded passwords. Based on smell density, we observed the most dominant and least dominant security smell to be respectively, 'Hard-coded secret' and 'Admin by default'. We observed security anti-patterns to persist; for example, hard-coded secrets can reside in an IaC script for up-to 98 months.

We have discussed the limitations of the thesis, which also provides opportunity for future researchers to build upon our work. We hope our dissertation will facilitate further security-related research in the domain of IaC scripts.

BIBLIOGRAPHY

[Aca17a] Acar, Y. et al. “Comparing the Usability of Cryptographic APIs”. *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 154–171. DOI: 10.1109/SP.2017.52.

[Aca17b] Acar, Y. et al. “Developers Need Support, Too: A Survey of Security Advice for Software Developers”. *2017 IEEE Cybersecurity Development (SecDev)*. 2017, pp. 22–26. DOI: 10.1109/SecDev.2017.17.

[AM16] Adams, B. & McIntosh, S. “Modern Release Engineering in a Nutshell – Why Researchers Should Care”. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. 2016, pp. 78–90. DOI: 10.1109/SANER.2016.108.

[Agr18] Agrawal, A. et al. “We Don’T Need Another Hero?: The Impact of "Heroes" on Software Development”. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’18. Gothenburg, Sweden: ACM, 2018, pp. 245–253. DOI: 10.1145/3183519.3183549. URL: http://doi.acm.org/10.1145/3183519.3183549.

[Ala08] Alali, A. et al. “What’s a Typical Commit? A Characterization of Open Source Software Repositories”. *2008 16th IEEE International Conference on Program Comprehension*. 2008, pp. 182–191. DOI: 10.1109/ICPC.2008.24.

[AF17] Alégroth, E. & Feldt, R. “On the long-term use of visual gui testing in industrial practice: a case study”. *Empirical Software Engineering* **22**.6 (2017), pp. 2937–2971. DOI: 10.1007/s10664-016-9497-6. URL: https://doi.org/10.1007/s10664-016-9497-6.

[AM05] Alhazmi, O. H. & Malaiya, Y. K. “Quantitative vulnerability assessment of systems software”. *Annual Reliability and Maintainability Symposium, 2005. Proceedings.* 2005, pp. 615–620. DOI: 10.1109/RAMS.2005.1408432.

[And02] Anderson, E. A. “Researching System Administration”. AAI3063285. PhD thesis. 2002.

[Arn16] Arnaoudova, V. et al. “Linguistic antipatterns: what they are and how developers perceive them”. *Empirical Software Engineering* **21**.1 (2016), pp. 104–158. DOI: 10.1007/s10664-014-9350-8. URL: https://doi.org/10.1007/s10664-014-9350-8.

[Bar16] Barker, E. *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*. Tech. rep. Gaithersburg, Maryland: National Institute of Standards and Technology, 2016, p. 81. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-175b.pdf.

[BP84] Basili, V. R. & Perricone, B. T. “Software errors and complexity: an empirical investigation0”. *Communications of the ACM* **27**.1 (1984), pp. 42–52.

[Bas09] Basso, T. et al. “An investigation of java faults operators derived from a field data study on java software faults”. *Workshop de Testes e Tolerância a Falhas*. 2009, pp. 1–13.

[Bat01] Battin, R. D. et al. “Leveraging resources in global software development”. *IEEE Software* **18**.2 (2001), pp. 70–77. DOI: 10.1109/52.914750.

[Bir11] Bird, C. et al. “Don’t Touch My Code!: Examining the Effects of Ownership on Software Quality”. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE ’11. Szeged, Hungary: ACM, 2011, pp. 4–14. DOI: 10.1145/2025113.2025119. URL: http://doi.acm.org/10.1145/2025113.2025119.

[Ble03] Blei, D. M. et al. “Latent dirichlet allocation”. *Journal of machine Learning research* **3**.Jan (2003), pp. 993–1022.

[BB94] Boer, B. den & Bosselaers, A. “Collisions for the Compression Function of MD5”. *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology*. EUROCRYPT ’93. Lofthus, Norway: Springer-Verlag New York, Inc., 1994, pp. 293–304. URL: http://dl.acm.org/citation.cfm?id=188307.188356.

[Bos14] Bosu, A. et al. “Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study”. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 257–268. DOI: 10.1145/2635868.2635880. URL: http://doi.acm.org/10.1145/2635868.2635880.

[Bre01] Breiman, L. “Random Forests”. *Machine Learning* **45**.1 (2001), pp. 5–32. DOI: 10.1023/A:1010933404324. URL: http://dx.doi.org/10.1023/A:1010933404324.

[Bre84] Breiman, L. et al. *Classification and Regression Trees*. 1st. New York: Chapman & Hall, 1984, p. 358. URL: http://www.crcpress.com/catalog/C4841.htm.

[Bri17] Bright, J. *Slalom’s approach to breaking down silos between DevOps and Security Teams*. https://blog.chef.io/2017/08/16/slaloms-approach-to-breaking-down-silos-between-devops-and-security/. [Online; accessed 18-Feb-2019]. 2017.

[Bro95] Brooks Jr., F. P. *The Mythical Man-month (Anniversary Ed.)* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[Bro98a] Brown, W. H. et al. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 1998.

[Bro98b] Brown, W. H. et al. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 1998.

[Bus17] Businge, J. et al. “Code Authorship and Fault-proneness of Open-Source Android Applications: An Empirical Study”. *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE. Toronto, Canada: ACM, 2017, pp. 33–42. DOI: 10.1145/3127005.3127009. URL: http://doi.acm.org/10.1145/3127005.3127009.

[But02] Butcher, M. et al. “Improving software testing via ODC: Three case studies”. *IBM Systems Journal* **41**.1 (2002), pp. 31–44. DOI: 10.1147/sj.411.0031.

[Cha14] Charmaz, K. *Constructing grounded theory*. London, UK: Sage Publishing, 2014.

[CJ17] Chen, B. & Jiang, Z. M. “Characterizing and Detecting Anti-Patterns in the Logging Code”. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017, pp. 71–81. DOI: 10.1109/ICSE.2017.15.

[Chi92] Chillarege, R. et al. “Orthogonal defect classification-a concept for in-process measurements”. *IEEE Transactions on Software Engineering* **18**.11 (1992), pp. 943–956. DOI: 10.1109/32.177364.

[CC96] Christmansson, J. & Chillarege, R. “Generation of an error set that emulates software faults based on field data”. *Proceedings of Annual Symposium on Fault Tolerant Computing*. 1996, pp. 304–313. DOI: 10.1109/FTCS.1996.534615.

[Cin14] Cinque, M. et al. “Assessing Direct Monitoring Techniques to Analyze Failures of Critical Industrial Systems”. *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014, pp. 212–222. DOI: 10.1109/ISSRE.2014.30.

[Cit15a] Cito, J. et al. “The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud”. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 393–403. DOI: 10.1145/2786805.2786826. URL: http://doi.acm.org/10.1145/2786805.2786826.

[Cit15b] Cito, J. et al. “The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud”. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 393–403. DOI: 10.1145/2786805.2786826. URL: http://doi.acm.org/10.1145/2786805.2786826.

[Cli93] Cliff, N. “Dominance statistics: Ordinal analyses to answer ordinal questions.” *Psychological Bulletin* **114**.3 (1993), pp. 494–509.

[Coh60] Cohen, J. “A Coefficient of Agreement for Nominal Scales”. *Educational and Psychological Measurement* **20**.1 (1960), pp. 37–46. DOI: 10.1177/001316446002000104. eprint: http://dx.doi.org/10.1177/001316446002000104. URL: http://dx.doi.org/10.1177/001316446002000104.

[Cot13] Cotroneo, D. et al. “Testing Techniques Selection Based on ODC Fault Types and Software Metrics”. *J. Syst. Softw.* **86**.6 (2013), pp. 1613–1637. DOI: 10.1016/j.jss.2013.02.020. URL: http://dx.doi.org/10.1016/j.jss.2013.02.020.

[CH04] Cramer, D. & Howitt, D. L. *The Sage dictionary of statistics: a practical resource for students in the social sciences*. Sage, 2004.

[CC12] Cryptography, L. of & (CrySyS), S. S. *sKyWIper (a.k.a. Flame a.k.a. Flamer): A complex malware for targeted attacks*. Tech. rep. Budapest, Hungary: Laboratory of Cryptography and System Security, 2012, p. 64. URL: http://www.crysys.hu/skywiper/skywiper.pdf.

[Cut07] Cutler, D. R. et al. “RANDOM FORESTS FOR CLASSIFICATION IN ECOLOGY”. *Ecology* **88**.11 (2007), pp. 2783–2792. DOI: 10.1890/07-0539.1. URL: http://dx.doi.org/10.1890/07-0539.1.

[Dev16] Devanbu, P. et al. “Belief and Evidence in Empirical Software Engineering”. *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: ACM, 2016, pp. 108–119. DOI: 10.1145/2884781.2884812. URL: http://doi.acm.org/10.1145/2884781.2884812.

[DM06] Duraes, J. A. & Madeira, H. S. “Emulation of Software Faults: A Field Data Study and a Practical Approach”. *IEEE Trans. Softw. Eng.* **32**.11 (2006), pp. 849–867. DOI: 10.1109/TSE.2006.113. URL: http://dx.doi.org/10.1109/TSE.2006.113.

[Ege13] Egele, M. et al. “An Empirical Study of Cryptographic Misuse in Android Applications”. *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’13. Berlin, Germany: ACM, 2013, pp. 73–84. DOI: 10.1145/2508859.2516693. URL: http://doi.acm.org/10.1145/2508859.2516693.

[EK13] El-Koka, A. et al. “Regularization parameter tuning optimization approach in logistic regression”. *2013 15th International Conference on Advanced Communications Technology (ICACT)*. 2013, pp. 13–18.

[EA17] Escobar-Avila, J. et al. “Text Retrieval-based Tagging of Software Engineering Video Tutorials”. *Proceedings of the 39th International Conference on Software Engineering Companion*. ICSE-C ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 341–343. DOI: 10.1109/ICSE-C.2017.121. URL: https://doi.org/10.1109/ICSE-C.2017.121.

[Fah12] Fahl, S. et al. “Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security”. *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA: ACM, 2012, pp. 50–61. DOI: 10.1145/2382196.2382205. URL: http://doi.acm.org/10.1145/2382196.2382205.

[Fel11] Felt, A. P. et al. “Android Permissions Demystified”. *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS ’11. Chicago, Illinois, USA: ACM, 2011, pp. 627–638. DOI: 10.1145/2046707.2046779. URL: http://doi.acm.org/10.1145/2046707.2046779.

[FN99] Fenton, N. E. & Neil, M. “Software metrics: successes, failures and new directions”. *Journal of Systems and Software* **47**.2 (1999), pp. 149 –157. DOI: https://doi.org/10.1016/S0164-1212(99)00035-7. URL: http://www.sciencedirect.com/science/article/pii/S0164121299000357.

[FO00] Fenton, N. E. & Ohlsson, N. “Quantitative Analysis of Faults and Failures in a Complex Software System”. *IEEE Trans. Softw. Eng.* **26**.8 (2000), pp. 797–814. DOI: 10.1109/32.879815. URL: http://dx.doi.org/10.1109/32.879815.

[Flo17] Florea, A.-C. et al. “Spark-Based Cluster Implementation of a Bug Report Assignment Recommender System”. *Artificial Intelligence and Soft Computing: 16th International Conference, ICAISC 2017, Zakopane, Poland, June 11-15, 2017, Proceedings, Part II*. Ed. by Rutkowski, L. et al. Cham: Springer International Publishing, 2017, pp. 31–42. DOI: 10.1007/978-3-319-59060-8_4. URL: https://doi.org\/10.1007\/978-3-319-59060-8_4.

[FV08] Fonseca, J. & Vieira, M. “Mapping software faults with web security vulnerabilities”. *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. 2008, pp. 257–266. DOI: 10.1109/DSN.2008.4630094.

[Fow99] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[Fow17] Fowler, M. *Feature Toggles (aka Feature Flags)*. https://martinfowler.com/articles/feature-toggles.html. [Online; accessed 08-Feb-2019]. 2017.

[FB99] Fowler, M. & Beck, K. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[FP16] Franceschi, A. & Pastor, J. S. *Extending Puppet - Second Edition*. 2nd. Packt Publishing, 2016.

[Fre05] Freedman, D. *Statistical Models : Theory and Practice*. Cambridge University Press, 2005.

[Fu16] Fu, W. et al. “Tuning for software analytics: Is it really necessary?” *Information and Software Technology* **76** (2016), pp. 135 –146. DOI: https://doi.org/10.1016/j.infsof.2016.04.017. URL: http://www.sciencedirect.com/science/article/pii/S0950584916300738.

[Gar91] Garland, R. “The mid-point on a rating scale: Is it desirable”. *Marketing Bulletin* (1991), pp. 66–70.

[Gen10] Genuer, R. et al. “Variable selection using random forests”. *Pattern Recognition Letters* **31**.14 (2010), pp. 2225 –2236. DOI: https://doi.org/10.1016/j.patrec.2010.03.014. URL: http://www.sciencedirect.com/science/article/pii/S0167865510000954.

[Gha17] Ghafari, M. et al. “Security Smells in Android”. *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2017, pp. 121–130. DOI: 10.1109/SCAM.2017.24.

[Gho15] Ghotra, B. et al. “Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models”. *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE ’15. Florence, Italy: IEEE Press, 2015, pp. 789–800. URL: http://dl.acm.org/citation.cfm?id=2818754.2818850.

[Gho17] Ghotra, B. et al. “A Large-scale Study of the Impact of Feature Selection Techniques on Defect Classification Models”. *Proceedings of the 14th International Conference on Mining Software Repositories*. MSR ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 146–157. DOI: 10.1109/MSR.2017.18. URL: https://doi.org/10.1109/MSR.2017.18.

[Gre15] Greiler, M. et al. “Code Ownership and Software Quality: A Replication Study”. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 2015, pp. 2–12. DOI: 10.1109/MSR.2015.8.

[GE03] Guyon, I. & Elisseeff, A. “An Introduction to Variable and Feature Selection”. *J. Mach. Learn. Res.* **3** (2003), pp. 1157–1182. URL: http://dl.acm.org/citation.cfm?id=944919.944968.

[Hal12] Hall, T. et al. “A Systematic Literature Review on Fault Prediction Performance in Software Engineering”. *IEEE Transactions on Software Engineering* **38**.6 (2012), pp. 1276–1304. DOI: 10.1109/TSE.2011.103.

[Hal14] Hall, T. et al. “Some Code Smells Have a Significant but Small Effect on Faults”. *ACM Trans. Softw. Eng. Methodol.* **23**.4 (2014), 33:1–33:39. DOI: 10.1145/2629648. URL: http://doi.acm.org/10.1145/2629648.

[Har54] Harris, Z. S. “Distributional Structure”. *WORD* **10**.2-3 (1954), pp. 146–162. DOI: 10.1080\/00437956.1954.11659520.

[Has09] Hassan, A. E. “Predicting Faults Using the Complexity of Code Changes”. *Proceedings of the 31st International Conference on Software Engineering*. ICSE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88. DOI: 10.1109/ICSE.2009.5070510. URL: http://dx.doi.org/10.1109/ICSE.2009.5070510.

[Hat10] Hata, H. et al. “Fault-prone module detection using large-scale text features based on spam filtering”. *Empirical Software Engineering* **15**.2 (2010), pp. 147–165. DOI: 10.1007/s10664-009-9117-9. URL: https://doi.org/10.1007/s10664-009-9117-9.

[Hat97] Hatton, L. “Reexamining the Fault Density-Component Size Connection”. *IEEE Softw.* **14**.2 (1997), pp. 89–97. DOI: 10.1109/52.582978. URL: http://dx.doi.org/10.1109/52.582978.

[HW04] Henningsson, K. & Wohlin, C. “Assuring Fault Classification Agreement " An Empirical Evaluation”. *Proceedings of the 2004 International Symposium on Empirical Software Engineering*. ISESE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 95–104. DOI: 10.1109/ISESE.2004.13. URL: http://dx.doi.org/10.1109/ISESE.2004.13.

[HA05] Hove, S. E. & Anda, B. “Experiences from conducting semi-structured interviews in empirical software engineering research”. *11th IEEE International Software Metrics Symposium (METRICS’05)*. 2005, 10 pp.–23. DOI: 10.1109/METRICS.2005.24.

[Hov12] Hovsepyan, A. et al. “Software Vulnerability Prediction Using Text Analysis Techniques”. *Proceedings of the 4th International Workshop on Security Measurements and Metrics*. MetriSec ’12. Lund, Sweden: ACM, 2012, pp. 7–10. DOI: 10.1145/2372225.2372230. URL: http://doi.acm.org/10.1145/2372225.2372230.

[Hud98] Hudak, P. “Modular domain specific languages and tools”. *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*. 1998, pp. 134–142. DOI: 10.1109/ICSR.1998.685738.

[HF10] Humble, J. & Farley, D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. 1st. Addison-Wesley Professional, 2010.

[Hum13] Hummer, W. et al. “Automated Testing of Chef Automation Scripts”. *Proceedings Demo:38; Poster Track of ACM/IFIP/USENIX International Middleware Conference*. MiddlewareDPT ’13. Beijing, China: ACM, 2013, 4:1–4:2. DOI: 10.1145/2541614.2541632. URL: http://doi.acm.org/10.1145/2541614.2541632.

[IEE10] IEEE. “IEEE Standard Classification for Software Anomalies”. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* (2010), pp. 1–23. DOI: 10.1109/IEEESTD.2010.5399061.

[JA15] Jiang, Y. & Adams, B. “Co-evolution of Infrastructure and Source Code: An Empirical Study”. *Proceedings of the 12th Working Conference on Mining Software Repositories*. MSR ’15. Florence, Italy: IEEE Press, 2015, pp. 45–55. URL: http://dl.acm.org/citation.cfm?id=2820518.2820527.

[Kel12] Keller, G. *Incident documentation20160204-Phabricator*. https://jumpcloud.com\/blog\/why-user-management-in-chef-and-puppet-is-a-mistake/. Blog. [Online; accessed 10-October-2017]. 2012.

[Kel92] Kelly, J. C. et al. “An analysis of defect densities found during software inspections”. *Journal of Systems and Software* **17**.2 (1992), pp. 111 –117. DOI: https://doi.org/10.1016/0164-1212(92)90089-3. URL: http://www.sciencedirect.com/science/article/pii/0164121292900893.

[Kim16] Kim, G. et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, 2016.

[Kis11] Kissel, R. *Glossary of key information security terms*. Diane Publishing, 2011.

[KP08] Kitchenham, B. A. & Pfleeger, S. L. “Personal Opinion Surveys”. *Guide to Advanced Empirical Software Engineering*. Ed. by Shull, F. et al. London: Springer London, 2008, pp. 63–92. DOI: 10.1007/978-1-84800-044-5_3. URL: https://doi.org/10.1007/978-1-84800-044-5_3.

[Kri16] Krishna, R. et al. “The "BigSE" Project: Lessons Learned from Validating Industrial Text Mining”. *Proceedings of the 2Nd International Workshop on BIG Data Software Engineering*. BIGDSE ’16. Austin, Texas: ACM, 2016, pp. 65–71. DOI: 10.1145/2896825.2896836. URL: http://doi.acm.org/10.1145/2896825.2896836.

[Kri18] Krishna, R. et al. “What is the Connection Between Issues, Bugs, and Enhancements?: Lessons Learned from 800+ Software Projects”. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’18. Gothenburg, Sweden: ACM, 2018, pp. 306–315. DOI: 10.1145/3183519.3183548. URL: http://doi.acm.org/10.1145/3183519.3183548.

[Lab17] Labs, P. *Puppet Documentation*. https://docs.puppet.com/. [Online; accessed 19-July-2017]. 2017.

[LK77] Landis, J. R. & Koch, G. G. “The Measurement of Observer Agreement for Categorical Data”. *Biometrics* **33**.1 (1977), pp. 159–174. URL: http://www.jstor.org/stable/2529310.

[Les08] Lessmann, S. et al. “Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings”. *IEEE Trans. Softw. Eng.* **34**.4 (2008), pp. 485–496. DOI: 10.1109/TSE.2008.35. URL: http://dx.doi.org/10.1109/TSE.2008.35.

[LC03] Levy, D. & Chillarege, R. “Early warning of failures through alarm analysis a case study in telecom voice mail systems”. *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* 2003, pp. 271–280. DOI: 10.1109/ISSRE.2003.1251049.

[Lew13] Lewis, C. et al. “Does Bug Prediction Support Human Developers? Findings from a Google Case Study”. *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 372–381. URL: http://dl.acm.org/citation.cfm?id=2486788.2486838.

[LM04] Lutz, R. R. & Mikulski, I. C. “Empirical analysis of safety-critical anomalies during operations”. *IEEE Transactions on Software Engineering* **30**.3 (2004), pp. 172–180. DOI: 10.1109/TSE.2004.1271171.

[Lyu03] Lyu, M. R. et al. “An empirical study on testing and fault tolerance for software reliability engineering”. *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* 2003, pp. 119–130. DOI: 10.1109/ISSRE.2003.1251036.

[Mac18] MacLeod, L. et al. “Code Reviewing in the Trenches: Challenges and Best Practices”. *IEEE Software* **35**.4 (2018), pp. 34–42. DOI: 10.1109/MS.2017.265100500.

[MW47] Mann, H. B. & Whitney, D. R. “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other”. *The Annals of Mathematical Statistics* **18**.1 (1947), pp. 50–60. URL: http://www.jstor.org/stable/2236101.

[Man08] Manning, C. D. et al. *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[ML06] Mantyla, M. V. & Lassenius, C. “Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study”. *Empirical Softw. Engg.* **11**.3 (2006), pp. 395–431. DOI: 10.1007/s10664-006-9002-8. URL: http://dx.doi.org/10.1007/s10664-006-9002-8.

[Mat10] Matsumoto, S. et al. “An Analysis of Developer Metrics for Fault Prediction”. *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. PROMISE ’10. Timi§oara, Romania: ACM, 2010, 18:1–18:9. DOI: 10.1145/1868328.1868356. URL: http://doi.acm.org/10.1145/1868328.1868356.

[MW03] Maximilien, E. M. & Williams, L. “Assessing Test-driven Development at IBM”. *Proceedings of the 25th International Conference on Software Engineering*. ICSE ’03. Portland, Oregon: IEEE Computer Society, 2003, pp. 564–569. URL: http://dl.acm.org/citation.cfm?id=776816.776892.

[MN98] McCallum, A. & Nigam, K. “A Comparison of Event Models for Naive Bayes Text Classification”. *Learning for Text Categorization: Papers from the 1998 AAAI Workshop*. 1998, pp. 41–48. URL: http://www.kamalnigam.com/papers/multinomial-aaaiws98.pdf.

[MJ11] McCune, J. T. & Jeffrey. *Pro Puppet*. 1st ed. Apress, 2011, p. 336. DOI: 10.1007/978-1-4302-3058-8. URL: https://www.springer.com/gp/book/9781430230571.

[MW09] Meneely, A. & Williams, L. “Secure Open Source Collaboration: An Empirical Study of Linus’ Law”. *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS ’09. Chicago, Illinois, USA: ACM, 2009, pp. 453–462. DOI: 10.1145/1653662.1653717. URL: http://doi.acm.org/10.1145/1653662.1653717.

[Men11] Meneely, A. et al. “Does Adding Manpower Also Affect Quality?: An Empirical, Longitudinal Analysis”. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE ’11. Szeged, Hungary: ACM, 2011, pp. 81–90. DOI: 10.1145/2025113.2025128. URL: http://doi.acm.org/10.1145/2025113.2025128.

[Men13] Meneely, A. et al. “Validating Software Metrics: A Spectrum of Philosophies”. *ACM Trans. Softw. Eng. Methodol.* **21**.4 (2013), 24:1–24:28. DOI: 10.1145/2377656.2377661. URL: http://doi.acm.org/10.1145/2377656.2377661.

[Men18] Meng, N. et al. “Secure Coding Practices in Java: Challenges and Vulnerabilities”. *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: ACM, 2018, pp. 372–383. DOI: 10.1145/3180155.3180201. URL: http://doi.acm.org/10.1145/3180155.3180201.

[Men07a] Menzies, T. et al. “Data Mining Static Code Attributes to Learn Defect Predictors”. *IEEE Transactions on Software Engineering* **33**.1 (2007), pp. 2–13. DOI: 10.1109/TSE.2007.256941.

[Men07b] Menzies, T. et al. “Problems with Precision: A Response to "Comments on ’Data Mining Static Code Attributes to Learn Defect Predictors’"”. *IEEE Trans. Softw. Eng.* **33**.9 (2007), pp. 637–640. DOI: 10.1109/TSE.2007.70721. URL: http://dx.doi.org/10.1109/TSE.2007.70721.

[Mik13] Mikolov, T. et al. “Distributed Representations of Words and Phrases and Their Compositionality”. *Proceedings of the 26th International Conference on Neural Information Processing Systems*. NIPS’13. Lake Tahoe, Nevada: Curran Associates Inc., 2013, pp. 3111–3119. URL: http://dl.acm.org/citation.cfm?id=2999792.2999959.

[MIT18] MITRE. *CWE-Common Weakness Enumeration*. https://cwe.mitre.org/index.html. [Online; accessed 08-Aug-2018]. 2018.

[Miz07] Mizuno, O. et al. “Spam Filter Based Approach for Finding Fault-Prone Software Modules”. *Fourth International Workshop on Mining Software Repositories (MSR’07:ICSE Workshops 2007)*. 2007, pp. 4–4. DOI: 10.1109/MSR.2007.29.

[Moh06] Moha, N. et al. “Automatic Generation of Detection Algorithms for Design Defects”. *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*. 2006, pp. 297–300. DOI: 10.1109/ASE.2006.22.

[Moh10] Moha, N. et al. “DECOR: A Method for the Specification and Detection of Code and Design Smells”. *IEEE Transactions on Software Engineering* **36**.1 (2010), pp. 20–36. DOI: 10.1109/TSE.2009.50.

[Moh04] Mohagheghi, P. et al. "An Empirical Study of Software Reuse vs. Defect-Density and Stability". *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 282–292. URL: http://dl.acm.org/citation.cfm?id=998675.999433.

[MP93] Moller, K. H. & Paulish, D. J. "An empirical investigation of software fault distribution". *[1993] Proceedings First International Software Metrics Symposium*. 1993, pp. 82–90. DOI: 10.1109/METRIC.1993.263798.

[Mor16] Morris, K. *Infrastructure as code: managing servers in the cloud*. " O'Reilly Media, Inc.", 2016.

[Mun17] Munaiah, N. et al. "Curating GitHub for engineered software projects". *Empirical Software Engineering* (2017), pp. 1–35. DOI: 10.1007/s10664-017-9512-6. URL: http://dx.doi.org/10.1007/s10664-017-9512-6.

[Mut99] Mutaf, P. "Defending against a Denial-of-Service Attack on TCP." *Recent Advances in Intrusion Detection*. 1999.

[NB05] Nagappan, N. & Ball, T. "Use of Relative Code Churn Measures to Predict System Defect Density". *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: ACM, 2005, pp. 284–292. DOI: 10.1145/1062455.1062514. URL: http://doi.acm.org/10.1145/1062455.1062514.

[Nag06] Nagappan, N. et al. "Mining Metrics to Predict Component Failures". *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, 2006, pp. 452–461. DOI: 10.1145/1134285.1134349. URL: http://doi.acm.org/10.1145/1134285.1134349.

[Nag08] Nagappan, N. et al. "The Influence of Organizational Structure on Software Quality: An Empirical Case Study". *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: ACM, 2008, pp. 521–530. DOI: 10.1145/1368088.1368160. URL: http://doi.acm.org/10.1145/1368088.1368160.

[Nat14] National Institute of Standards and Technology. *Security and Privacy Controls for Federal Information Systems and Organizations*. https://www.nist.gov/publications/security-and-privacy-controls-federal-information-systems-and-organizations-including-0. [Online; accessed 18-Aug-2018]. 2014.

[Nat18] National Institute of Standards and Technology. *CSRC-Glossary-Vulnerability*. https://csrc.nist.gov/Glossary/?term=2436. [Online; accessed 09-Aug-2018]. 2018.

[NV17] Nunez-Varela, A. S. et al. "Source code metrics: A systematic mapping study". *Journal of Systems and Software* **128** (2017), pp. 164 –197. DOI: https://doi.org/10.1016/j.jss.2017.03.044. URL: http://www.sciencedirect.com/science/article/pii/S0164121217300663.

[Pal14] Palomba, F. et al. “Do They Really Smell Bad? A Study on Developers’ Perception of Bad Code Smells”. *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. ICSME ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 101–110. DOI: 10.1109/ICSME.2014.32. URL: http://dx.doi.org/10.1109/ICSME.2014.32.

[Pan96] Pandit, N. R. “The creation of theory: A recent application of the grounded theory method”. *The Qualitative Report* **2**.4 (1996), pp. 1–20. URL: https://ueaeprints.uea.ac.uk/28591/.

[Par17] Parnin, C. et al. “The Top 10 Adages in Continuous Deployment”. *IEEE Software* **34**.3 (2017), pp. 86–95. DOI: 10.1109/MS.2017.86.

[PR12] Pecchia, A. & Russo, S. “Detection of Software Failures through Event Logs: An Experimental Study”. *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. 2012, pp. 31–40. DOI: 10.1109/ISSRE.2012.24.

[Ped11] Pedregosa, F. et al. “Scikit-learn: Machine Learning in Python”. *J. Mach. Learn. Res.* **12** (2011), pp. 2825–2830. URL: http://dl.acm.org/citation.cfm?id=1953048.2078195.

[Per15] Perl, H. et al. “VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits”. *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS ’15. Denver, Colorado, USA: ACM, 2015, pp. 426–437. DOI: 10.1145/2810103.2813604. URL: http://doi.acm.org/10.1145/2810103.2813604.

[Pet08] Petersen, K. et al. “Systematic Mapping Studies in Software Engineering”. *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*. EASE’08. Italy: BCS Learning & Development Ltd., 2008, pp. 68–77. URL: http://dl.acm.org/citation.cfm?id=2227115.2227123.

[Pin08] Pinzger, M. et al. “Can Developer-module Networks Predict Failures?” *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT ’08/FSE-16. Atlanta, Georgia: ACM, 2008, pp. 2–12. DOI: 10.1145/1453101.1453105. URL: http://doi.acm.org/10.1145/1453101.1453105.

[Por97] Porter, M. F. “Readings in Information Retrieval”. Ed. by Sparck Jones, K. & Willett, P. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. Chap. An Algorithm for Suffix Stripping, pp. 313–316. URL: http://dl.acm.org/citation.cfm?id=275537.275705.

[RW18] Rahman, A. & Williams, L. “Characterizing Defective Configuration Scripts Used for Continuous Deployment”. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018, pp. 34–45. DOI: 10.1109/ICST.2018.00014.

[RW19] Rahman, A. & Williams, L. “Source Code Properties of Defective Infrastructure as Code Scripts”. *Information and Software Technology* (2019). DOI: https://doi.org/10.1016/j.infsof.2019.04.013. URL: http://www.sciencedirect.com/science/article/pii/S0950584919300965.

[Rah17] Rahman, A. et al. “Predicting Android Application Security and Privacy Risk with Static Code Metrics”. *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 2017, pp. 149–153. DOI: 10.1109/MOBILESoft.2017.14.

[Rah18a] Rahman, A. et al. “A systematic mapping study of infrastructure as code research”. *Information and Software Technology* (2018). DOI: https://doi.org/10.1016/j.infsof.2018.12.004. URL: http://www.sciencedirect.com/science/article/pii/S0950584918302507.

[Rah18b] Rahman, A. et al. “Categorizing Defects in Infrastructure as Code”. *CoRR* **abs/1809.07937** (2018). arXiv: 1809.07937. URL: http://arxiv.org/abs/1809.07937.

[Rah18c] Rahman, A. et al. “Characterizing the Influence of Continuous Integration: Empirical Results from 250+ Open Source and Proprietary Projects”. *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*. SWAN 2018. Lake Buena Vista, FL, USA: ACM, 2018, pp. 8–14. DOI: 10.1145/3278142.3278149. URL: http://doi.acm.org/10.1145/3278142.3278149.

[Rah19a] Rahman, A. et al. “Snakes in Paradise?: Insecure Python-related Coding Practices in Stack Overflow”. *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR ’19. Montreal, Canada, 2019.

[Rah19b] Rahman, A. et al. “The Seven Sins: Security Smells in Infrastructure as Code Scripts”. *Proceedings of the 41st International Conference on Software Engineering*. ICSE ’19. To appear. Pre-print: https://akondrahman.github.io/papers/icse19_slic.pdf. Montreal, Canada, 2019.

[Rah15] Rahman, A. A. U. et al. “Synthesizing Continuous Deployment Practices Used in Software Development”. *Proceedings of the 2015 Agile Conference*. AGILE ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 1–10. DOI: 10.1109/Agile.2015.12. URL: http://dx.doi.org/10.1109/Agile.2015.12.

[RD13] Rahman, F. & Devanbu, P. “How, and Why, Process Metrics Are Better”. *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 432–441. URL: http://dl.acm.org/citation.cfm?id=2486788.2486846.

[Rah14] Rahman, F. et al. “Comparing Static Bug Finders and Statistical Prediction”. *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 424–434. DOI: 10.1145/2568225.2568269. URL: http://doi.acm.org/10.1145/2568225.2568269.

[Ray16] Ray, B. et al. “On the "Naturalness" of Buggy Code”. *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: ACM, 2016, pp. 428–439. DOI: 10.1145/2884781.2884848. URL: http://doi.acm.org/10.1145/2884781.2884848.

[Res00] Rescorla, E. “Http over tls” (2000).

[Rom06] Romano, J. et al. “Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen’sd for evaluating group differences on the NSSE and other surveys?” *annual meeting of the Florida Association of Institutional Research*. 2006, pp. 1–3.

[Sal15] Saldaña, J. *The coding manual for qualitative researchers*. Sage, 2015.

[SS75] Saltzer, J. H. & Schroeder, M. D. “The protection of information in computer systems”. *Proceedings of the IEEE* **63**.9 (1975), pp. 1278–1308. DOI: 10.1109/PROC.1975.9939.

[Sca14] Scandariato, R. et al. “Predicting Vulnerable Software Components via Text Mining”. *IEEE Transactions on Software Engineering* **40**.10 (2014), pp. 993–1006. DOI: 10.1109/TSE.2014.2340398.

[Sha16a] Shambaugh, R. et al. “Rehearsal: A Configuration Verification Tool for Puppet”. *SIGPLAN Not.* **51**.6 (2016), pp. 416–430. DOI: 10.1145/2980983.2908083. URL: http://doi.acm.org/10.1145/2980983.2908083.

[Sha16b] Sharma, T. et al. “Does Your Configuration Code Smell?” *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR ’16. Austin, Texas: ACM, 2016, pp. 189–200. DOI: 10.1145/2901739.2901761. URL: http://doi.acm.org/10.1145/2901739.2901761.

[Sli05] Sliwerski, J. et al. “When Do Changes Induce Fixes?” *Proceedings of the 2005 International Workshop on Mining Software Repositories*. MSR ’05. St. Louis, Missouri: ACM, 2005, pp. 1–5. DOI: 10.1145/1082983.1083147. URL: http://doi.acm.org/10.1145/1082983.1083147.

[Smi13] Smith, E. et al. “Improving developer participation rates in surveys”. *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 2013, pp. 89–92. DOI: 10.1109/CHASE.2013.6614738.

[Sto16] Stol, K.-J. et al. “Grounded Theory in Software Engineering Research: A Critical Review and Guidelines”. *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: ACM, 2016, pp. 120–131. DOI: 10.1145/2884781.2884833. URL: http://doi.acm.org/10.1145/2884781.2884833.

[Sto08] Storey, M.-A. et al. “TODO or to Bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers”. *Proceedings of the 30th International Conference on Software Engineering*. ICSE ’08. Leipzig, Germany: ACM, 2008, pp. 251–260. DOI: 10.1145/1368088.1368123. URL: http://doi.acm.org/10.1145/1368088.1368123.

[SP97] Storn, R. & Price, K. “Differential Evolution &Ndash; A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces”. *J. of Global Optimization* **11**.4 (1997), pp. 341–359. DOI: 10.1023/A:1008202821328. URL: http://dx.doi.org/10.1023/A:1008202821328.

[SC94] Strauss, A. & Corbin, J. “Grounded theory methodology”. *Handbook of qualitative research* **17** (1994), pp. 273–85.

[SF12] Sullivan, G. M. & Feinn, R. “Using Effect Size-or Why the P Value Is Not Enough”. *Journal of Graduate Medical Education* **4**.3 (2012), pp. 279–282. DOI: 10.4300/JGME-D-12-00156.1. eprint: https://doi.org/10.4300/JGME-D-12-00156.1. URL: https://doi.org/10.4300/JGME-D-12-00156.1.

[SC91] Sullivan, M. & Chillarege, R. “Software defects and their impact on system availability-a study of field failures in operating systems”. *[1991] Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium*. 1991, pp. 2–9. DOI: 10.1109/FTCS.1991.146625.

[Tan07] Tan, L. et al. “/*Icomment: Bugs or Bad Comments?*/”. *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP ’07. Stevenson, Washington, USA: ACM, 2007, pp. 145–158. DOI: 10.1145/1294261.1294276. URL: http://doi.acm.org/10.1145/1294261.1294276.

[Tan05] Tan, P.-N. et al. *Introduction to Data Mining, (First Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[Tan16] Tantithamthavorn, C. et al. “Automated Parameter Optimization of Classification Techniques for Defect Prediction Models”. *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: ACM, 2016, pp. 321–332. DOI: 10.1145/2884781.2884857. URL: http://doi.acm.org/10.1145/2884781.2884857.

[Tan17] Tantithamthavorn, C. et al. “An Empirical Comparison of Model Validation Techniques for Defect Prediction Models”. *IEEE Trans. Softw. Eng.* **43**.1 (2017), pp. 1–18. DOI: 10.1109/TSE.2016.2584050. URL: https://doi.org/10.1109/TSE.2016.2584050.

[Thu12] Thung, F. et al. “An Empirical Study of Bugs in Machine Learning Systems”. *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. 2012, pp. 271–280. DOI: 10.1109/ISSRE.2012.22.

[Tuf17] Tufano, M. et al. “An empirical study on developer-related factors characterizing fix-inducing commits”. *Journal of Software: Evolution and Process* **29**.1 (2017). e1797 JSME-15-0185.R2, e1797. DOI: 10.1002/smr.1797. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1797. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1797.

[Tur07] Turnbull, J. *Pulling Strings with Puppet. Configuration Management Made Easy*. 1st. Apress, 2007.

[URW16] Ur Rahman, A. A. & Williams, L. “Software Security in DevOps: Synthesizing Practitioners’ Perceptions and Practices”. *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*. CSED ’16. Austin, Texas: ACM, 2016, pp. 70–76. DOI: 10.1145/2896941.2896946. URL: http://doi.acm.org/10.1145/2896941.2896946.

[VW07] Van Wyk, E. et al. “Attribute Grammar-based Language Extensions for Java”. *Proceedings of the 21st European Conference on Object-Oriented Programming*. ECOOP’07. Berlin, Germany: Springer-Verlag, 2007, pp. 575–599. URL: http://dl.acm.org/citation.cfm?id=2394758.2394796.

[Voe13] Voelter, M. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. USA: CreateSpace Independent Publishing Platform, 2013.

[Wal14] Walden, J. et al. “Predicting Vulnerable Components: Software Metrics vs Text Mining”. *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014, pp. 23–33. DOI: 10.1109/ISSRE.2014.32.

[WY05] Wang, X. & Yu, H. “How to Break MD5 and Other Hash Functions”. *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT’05. Aarhus, Denmark: Springer-Verlag, 2005, pp. 19–35. DOI: 10.1007/11426639_2. URL: http://dx.doi.org/10.1007/11426639_2.

[Wei92] Weinberg, G. M. *Quality Software Management (Vol. 1): Systems Thinking*. New York, NY, USA: Dorset House Publishing Co., Inc., 1992.

[Wei17] Weiss, A. et al. “Tortoise: Interactive System Configuration Repair”. *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 625–636. URL: http://dl.acm.org/citation.cfm?id=3155562.3155641.

[YL06] Ylonen, T. & Lonvick, C. “The secure shell (SSH) protocol architecture” (2006).

[Zha17] Zhang, F. et al. “The Use of Summation to Aggregate Software Metrics Hinders the Performance of Defect Prediction Models”. *IEEE Transactions on Software Engineering* **43**.5 (2017), pp. 476–491. DOI: 10.1109/TSE.2016.2599161.

[Zha16] Zhang, F. et al. “Towards Building a Universal Defect Prediction Model with Rank Transformed Predictors”. *Empirical Softw. Engg.* **21**.5 (2016), pp. 2107–2145. DOI: 10.1007/s10664-015-9396-2. URL: http://dx.doi.org/10.1007/s10664-015-9396-2.

[Zha04] Zhang, H. “The Optimality of Naive Bayes”. *In FLAIRS2004 conference*. 2004.

[Zhe06a] Zheng, J. et al. “On the value of static analysis for fault detection in software”. *IEEE Transactions on Software Engineering* **32**.4 (2006), pp. 240–253. DOI: 10.1109/TSE.2006.38.

[Zhe06b] Zheng, J. et al. “On the Value of Static Analysis for Fault Detection in Software”. *IEEE Trans. Softw. Eng.* **32**.4 (2006), pp. 240–253. DOI: 10.1109/TSE.2006.38. URL: http://dx.doi.org/10.1109/TSE.2006.38.

[Zim07] Zimmermann, T. et al. “Predicting Defects for Eclipse”. *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. PROMISE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 9–. DOI: 10.1109/PROMISE.2007.10. URL: http://dx.doi.org/10.1109/PROMISE.2007.10.

APPENDICES

1. Appendix

Table A1: List of 31 Publications for Systematic Mapping Study

Index	Publication
S1	Joel Scheuner, Jrgen Cito, Philipp Leitner, and Harald Gall. 2015. "Cloud WorkBench: Benchmarking IaaS Providers based on Infrastructure-as-Code". In Proceedings of the 24th International Conference on World Wide Web (WWW '15 Companion). ACM, New York, NY, USA, 239-242.
S2	Yujuan Jiang and Bram Adams. 2015. "Co-evolution of infrastructure and source code: an empirical study". In Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15). IEEE Press, Piscataway, NJ, USA, 45-55.
S3	M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero and D. A. Tamburri, "DevOps: Introducing Infrastructure-as-Code," In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, 2017.
S4	Ikeshita, K., Ishikawa, F., Honiden, S., Gabmeyer, S., Johnsen, E., "Test Suite Reduction in Idempotence Testing of Infrastructure as Code", In Proceedings of the International Conference on Tests and Proofs (TAP'17), 2017.
S5	Hummer, W., Rosenberg, F., Oliveira, F., Eilam, T., "Testing Idempotence for Infrastructure as Code", IIn Proceedings of the International Conference on Middleware (Middleware'13), 2013.
S6	Oliver H., Waldemar H., and Schahram D., "Asserting reliable convergence for configuration management scripts", SIGPLAN Not. 51, 10 (October 2016).
	Continued on next page

Table A1 – continued from previous page

Index	Publication
S7	Nishant Kumar Singh, S. Thakur, H. Chaurasiya and H. Nagdev, "Automated provisioning of application in IAAS cloud using Ansible configuration management," in Proceedings of the 2015 1st International Conference on Next Generation Computing Technologies (NGCT), Dehradun, 2015.
S8	T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?", In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). ACM, New York, NY, USA.
S9	J. Hintsch, C. Grling and K. Turowski, "Modularization of Software as a Service Products: A Case Study of the Configuration Management Tool Puppet," In Proceedings of the 2015 International Conference on Enterprise Systems (ES), Basel, 2015.
S10	Aaron Weiss, Arjun Guha, and Yuriy Brun, "Tortoise: interactive system configuration repair", In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). IEEE Press, Piscataway, NJ, USA.
S11	Waldemar Hummer, Florian Rosenberg, Fabio Oliveira, and Tamar Eilam. 2013. "Automated testing of chef automation scripts", In Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference (MiddlewareDPT '13). ACM, New York, NY, USA.
S12	Johannes Wettinger, Uwe Breitenbcher, and Frank Leymann. "Standards-Based DevOps Automation and Integration Using TOSCA", In Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC '14). IEEE Computer Society, Washington, DC, USA.

Continued on next page

Table A1 – continued from previous page

Index	Publication
S13	B. Adams and S. McIntosh, "Modern Release Engineering in a Nutshell – Why Researchers Should Care," In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Suita, 2016.
S14	D. Spinellis, "Don't Install Software by Hand," in IEEE Software, vol. 29, no. 4, pp. 86-87, July-Aug. 2012.
S15	M. Miglierina, "Application Deployment and Management in the Cloud," In Proceedings of the 2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, 2014.
S16	Kecskemeti G., Kertesz A., Marosi A.C., "Towards a Methodology to Form Microservices from Monolithic Ones", In Proceedings of the Euro-Par 2016: Parallel Processing Workshops. Euro-Par 2016, 2016.
S17	Wettinger, J., Breitenbcher, U., Leymann, F., "DevOpSlang Bridging the Gap between Development and Operations", In Service-Oriented and Cloud Computing, 2014.
S18	de Gouw, S., Lienhardt, M., Mauro, J., Nobakht, B., Zavattaro, G., "On the Integration of Automatic Deployment into the ABS Modeling Language", In Service Oriented and Cloud Computing, 2015.
S19	Johannes W., Uwe B., Oliver K., and Leymann F., "Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel", In Future Generation Computer Systems 56, C (March 2016).

Continued on next page

Table A1 – continued from previous page

Index	Publication
S20	Eelco Dolstra, Rob Vermaas, and Shea Levy, "Charon: declarative provisioning and deployment", In Proceedings of the 1st International Workshop on Release Engineering (RELENG '13). IEEE Press, Piscataway, NJ, USA.
S21	P. Kathiravelu and L. Veiga, "SENDIM for Incremental Development of Cloud Networks: Simulation, Emulation and Deployment Integration Middleware," In Proceedings of the 2016 IEEE International Conference on Cloud Engineering (IC2E), Berlin, 2016.
S22	T. Karvinen and S. Li, "Investigating survivability of configuration management tools in unreliable and hostile networks," In Proceedings of the 2017 3rd International Conference on Information Management (ICIM), Chengdu, 2017.
S23	Chris Parnin, Eric Helms, Chris Atlee, Harley Boughton, Mark Ghattas, Andy Glover, James Holman, John Micco, Brendan Murphy, Tony Savor, Michael Stumm, Shari Whitaker, and Laurie Williams, "The Top 10 Adages in Continuous Deployment", In IEEE Softw. 34, 3 (May 2017).
S24	Salman Baset, Sahil Suneja, Nilton Bila, Ozan Tuncer, and Canturk Isci, "Usable declarative configuration specification and validation for applications, systems, and cloud", In Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track (Middleware '17).
S25	Lwakatare, L., Kuvaja, P., Leymann, F., Oivo, M., "Dimensions of DevOps", In Proceedings of the Agile Processes in Software Engineering and Extreme Programming, 2015.
	Continued on next page

Table A1 – continued from previous page

Index	Publication
S26	J. Wettinger, V. Andrikopoulos and F. Leymann, "Automated Capturing and Systematic Usage of DevOps Knowledge for Cloud Applications," In Proceedings of the 2015 IEEE International Conference on Cloud Engineering, Tempe, AZ, 2015.
S27	Marco Miglierina and Damian A. Tamburri, "Towards Omnia: A Monitoring Factory for Quality-Aware DevOps", In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (ICPE '17 Companion). ACM, New York, NY, USA.
S28	M. Virmani, "Understanding DevOps & bridging the gap from continuous integration to continuous delivery," In Proceedings of the Fifth International Conference on the Innovative Computing Technology (INTECH 2015), Galcia, 2015.
S29	Aiftimiei C., Costantini A., Bucchi R., Italiano A., Michelotto D., Panella M., Pergolesi M., Saletta M., Traldi S., Vistoli C., Zizzi G., Salomoni D., "Cloud Environment Automation: from infrastructure deployment to application monitoring", In Journal of Physics: Conference Series, 2017.
S30	J. Sandobalin, E. Insfran and S. Abrahao, "An Infrastructure Modelling Tool for Cloud Provisioning," In Proceedings of the 2017 IEEE International Conference on Services Computing (SCC), Honolulu, HI, 2017.
S31	Sandobalin, J., Insfran, E., Abrahao, S., "End-to-End Automation in Cloud Infrastructure Provisioning", In Proceedings of the Information Systems Development: Advances in Methods, Tools and Management Conference (ISD2017), Larnaca, Cyprus, 2017.

Continued on next page

Table A1 – continued from previous page

Index	Publication
S32	J. Scheuner, P. Leitner, J. Cito and H. Gall, “Cloud Work Bench – Infrastructure-as-Code Based Cloud Benchmarking,” In Proceedings of the 2014 IEEE 6th International Conference on Cloud Computing Technology and Science, Singapore, 2014.