Characterizing Defective Configuration Scripts Used for Continuous Deployment

Akond Rahman*, Laurie Williams†
North Carolina State University, Raleigh, North Carolina
Email: *aarahman@ncsu.edu, †williams@csc.ncsu.edu

Abstract—**In software engineering, validation and verification (V&V) resources are limited and characterization of defective software source files can help in efficiently allocating V&V resources. Similar to software source files, defects occur in the scripts used to automatically manage configurations and software deployment infrastructure, often known as infrastructure as code (IaC) scripts. Defects in IaC scripts can have dire consequences, for example, creating large-scale system outages. Identifying the characteristics of defective IaC scripts can help in mitigating these defects by allocating V&V efforts efficiently based upon these characteristics.** ***The objective of this paper is to help software practitioners to prioritize validation and verification efforts for infrastructure as code (IaC) scripts by identifying the characteristics of defective IaC scripts*****. Researchers have previously extracted text features to characterize defective software source files written in general purpose programming languages. We investigate if text features can be used to identify properties that characterize defective IaC scripts. We use two text mining techniques to extract text features from IaC scripts: the bag-of-words technique, and the term frequency-inverse document frequency (TF-IDF) technique. Using the extracted features and applying grounded theory, we characterize defective IaC scripts. We also use the text features to build defect prediction models with tuned statistical learners. We mine open source repositories from Mozilla, Openstack, and Wikimedia Commons, to construct three case studies and evaluate our methodology. We identify three properties that characterize defective IaC scripts: filesystem operations, infrastructure provisioning, and managing user accounts. Using the bag-of-word technique, we observe a median F-Measure of 0.74, 0.71, and 0.73, respectively, for Mozilla, Openstack, and Wikimedia Commons. Using the TF-IDF technique, we observe a median F-Measure of 0.72, 0.74, and 0.70, respectively, for Mozilla, Openstack, and Wikimedia Commons.**

I. Introduction

With respect to money, personnel, and time, validation and verification (V&V) of software source files is an expensive procedure, and V&V efforts should be allocated efficiently [1] [2] [3]. Previously, researchers [4] [5] [6] have investigated which characteristics are related to defective software source files. Using the identified characteristics, software practitioners can make informed decisions on prioritizing V&V efforts, by looking at software source files that contain the identified characteristics [7] [8].

Continuous deployment (CD) is a software development methodology that helps information technology (IT) organizations to deploy software rapidly [9] [10]. In CD, configurations and infrastructure specifics of the deployment environment is treated as code in form of scripts, known as infrastructure as code (IaC) scripts [11] [12]. Similar to software source files, practitioners change IaC scripts frequently [12] [13], and inadvertently introduce defects [12]. Defects in IaC scripts can have serious consequences, for example, in 2016 the Phabricator service of Wikimedia Commons became unavailable due to an erroneous configuration in an IaC script [14]. Through qualitative and quantitative analysis we can identify properties that characterize defective IaC scripts. Such characterization can help IT organizations to efficiently allocate their V&V efforts by focusing on scripts that contain the identified properties.

In prior work, researchers have used text features to characterize defective software source files written in general purpose programming languages (GPLs), such as Java [15] [5]. IaC scripts use domain specific languages (DSLs) [16]. The syntax and semantics of DSLs are fundamentally different from GPLs [17] [18] [19], and through systematic investigation we can determine if text-based features can be used effectively for characterizing and predicting defective IaC scripts.

The objective of this paper is to help software practitioners to prioritize validation and verification efforts for infrastructure as code (IaC) scripts by identifying the characteristics of defective IaC scripts.

We answer the following research questions:

RQ-1: Which are the characteristics of defective infrastructure as code (IaC) scripts? How frequently do the identified characteristics appear in IaC scripts?

RQ-2: How can we build prediction models for defective infrastructure as code scripts using text features?

We characterize defective IaC scripts by extracting text features. We use two text mining techniques to extract text features: the 'bag-of-words (BOW)' technique [20] and the 'term frequency-inverse document frequency (TF-IDF)' technique [21]. We apply feature selection [22] on the extracted features using principal component analysis (PCA) to account for collinearity and identify text features that are more correlated with defective IaC scripts. We apply the Strauss-Corbin Grounded Theory (SGT) [23] on text features that correlate with defective IaC scripts to characterize properties of defective IaC scripts. We quantify the count of each identified property that appear in IaC scripts. We construct defect prediction models using the text features and Random Forest (RF) [24]. We also tune the parameters of RF, to achieve better prediction performance. We evaluate the performance of the constructed prediction models using two metrics: area under the receiver operating characteristic curve (AUC) and F-Measure. We evaluate our methodology by constructing three datasets collected from three organizations namely, Mozilla, Openstack, and Wikimedia Commons. We mine open source repositories from these organizations to construct the three datasets. The count of IaC scripts is 580, 1383, and 296 respectively, for Mozilla, Openstack, and Wikimedia Commons.

We list our contributions as following:

- A list of properties that characterizes defective IaC scripts
- Defect prediction models that predict defective IaC scripts

We organize rest of the paper as following: in Section II we provide necessary background and related work. In Section III we describe our methodology. In Section IV we describe our case studies. We use Sections V and VI respectively, to report our empirical findings, and discuss our findings. We present the limitations of our paper in Section VII. We finally conclude the paper in Section VIII.

II. Background and Related Work

In this section we briefly describe prior work and necessary background needed for the paper.

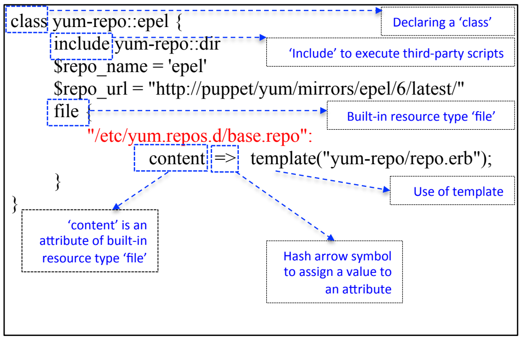
A. Related Work

Our paper is related to prior research that studied quality issues of IaC scripts. Sharma et al. [25] investigated smells in IaC scripts and proposed 13 implementation and 11 design configuration smells. Hanappi et al. [26] proposed an automated model-based testing framework for IaC scripts. Jiang and Adams [13] investigated the co-evolution of IaC scripts and other software artifacts, such as build files and source code. They observed that IaC scripts churn frequently, which can potentially introduce defects. The above-mentioned studies motivate us to study the characteristics of defective IaC scripts.

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. [5] mined token frequencies from 20 Android applications and used the mined text features to predict vulnerabilities that appear in these applications. Walden et al. [27] 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. [28] applied the BOW model on commit messages extracted from version control repositories to predict security defects in 66 Github projects. Hovsepyan et al. [29] 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. [30] 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. [15] 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. We use the text features to characterize defective IaC scripts.

B. Background

IaC script is the technology of automatically defining and managing computing and network configurations, and infrastructure through source code [11]. IaC scripts use domain specific language (DSL) [16]. DevOps organizations widely use commercial tools, such as Puppet to implement IaC [11] [13] [16]. IaC scripts are also known as configuration as code scripts [25] [11] or configuration scripts [31].



```
class yum-repo::epel {
      include yum-repo::dir
      $repo_name = 'epel'
      $repo_url = "http://puppet/yum/mirrors/epel/6/latest/"
      file {
           "/etc/yum.repos.d/base.repo":
                    content => template("yum-repo/repo.erb");
      }
}
```

Declaring a 'class'
'Include' to execute third-party scripts
Built-in resource type 'file'
Use of template
'content' is an attribute of built-in resource type 'file'
Hash arrow symbol to assign a value to an attribute

Fig. 1: Annotation of a Puppet script retrieved from Mozilla.

We provide background on Puppet scripts, because we use Puppet scripts to construct our case studies, as described in Section IV. Typical entities of Puppet scripts include modules, manifests, and resources [32]. A module is a collection of manifests. A manifest is composed of resources, and declarations of resource types. Manifests are written as scripts that use a .pp extension. As shown in Figure 1, a manifest script can contain a 'class' which is a named block of Puppet code. Classes in Puppet scripts are not related to OOP technology, and act as a placeholder for Puppet code elements. Each configuration item is referred as a 'resource'. 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 'file' can have an attribute called 'content' to specify which artifact will be used to provide necessary content. Content of files can be rendered using templates, as indicated by the 'template' method. Execution of external library or modules are also possible using the 'include' syntax. For better understanding, we provide a sample Puppet code retrieved from Mozilla [1] with annotations in Figure 1. The code snippet highlighted in red indicates a defect.

III. Methodology

We first provide definitions, then we describe our methodology.

- **Defect**: An imperfection in an IaC script that needs to be replaced or repaired. We follow the IEEE definition of defects [33].

[1] http://tiny.cc/bg-pupp

- **Defect-related commit**: A commit whose message indicates that an action was taken related to a defect.
- **Defective script**: An IaC script which is listed in a defect-related commit.

A. Dataset Construction

Our methodology of dataset construction involves three steps: repository collection (Section III-A1), commit message processing (Section III-A2), and determining defect-related commits (Section III-A3).

1) Repository Collection: Researchers [34] [35], in prior work on defect prediction, used datasets from public software data archives such as Tera-PROMISE and NASA. But these datasets are derived from OOP-based systems [36] [35], and not from IaC scripts. Thus, we need to construct IaC script-specific datasets to evaluate our methodology and build prediction models. We use open source repositories to construct our datasets. An open source repository contains valuable information about the development process of an open source project, but the project might have a short development period [37]. This observation motivates us to identify repositories for mining by using these criteria:

- **Criteria-1**: The repository must be available for download.
- **Criteria-2**: At least 11% of the files belonging to the repository must be IaC scripts. Jiang and Adams [13] reported that in open source repositories IaC scripts co-exist with other types of files, such as Makefiles and source code files. They observed a median of 11.1% 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. [37] used the threshold of at least two commits per month to determine which repositories have enough development activity for software organizations. We use this threshold to filter repositories that contain projects with short development activity.

2) Commit Message Processing: Prior research [38], [39] leveraged open source repositories that use version control systems (VCS) for defect prediction studies. We use two artifacts from VCS of the selected repositories from Section III-A1, 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 as following:

- 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 [40].
- 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 [40].
- 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 ‘combined commit message (COCM)’ throughout the rest of the paper. We use the extracted COCMs to separate the defect-related commits from the non defect-related commits, as described in III-A3.

3) Determining Defect-related Commits: We use defect-related commits to identify the defective IaC scripts, and the metrics that characterizes defective IaC scripts. We apply qualitative analysis to determine which commits were defect-related commits. Qualitative analysis provides the opportunity to improve the quality of the constructed dataset [41]. 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 determine a COCM as defect-related if the COCM represents an imperfection in an IaC script. We provide raters with an electronic handbook on IaC scripts [32], and the IEEE publication on anomaly classification [33]. We also record agreement between raters and the Cohen’s Kappa [42] score, for all COCMs.

Resolution Phase: Raters can disagree if a COCM 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 COCMs for each dataset, and contact practitioners. We ask the practitioners if they agree to our categorization of COCMs. High agreement between the raters’ categorization and programmers’ feedback is an indication of how well the raters performed. The percentage of COCMs to which practitioners agreed upon should be recorded and the Cohen’s Kappa score should be computed.

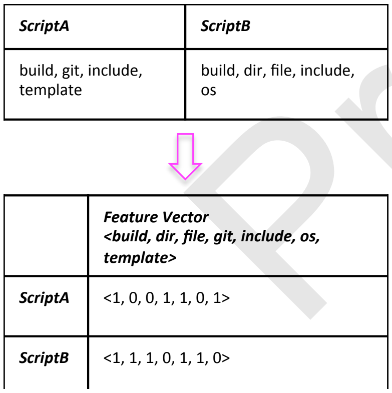
Upon completion of these three steps, we can classify which commits and COCMs are defect-related. From the defect-related commits we determine which IaC scripts are defective, similar to prior work [39]. Defect-related commits list which IaC scripts were changed, and from this list we determine which IaC scripts are defective. From the defective and non-defective scripts we extract text features using two steps: text preprocessing and text feature extraction, respectively discussed in Section III-A4 and Section III-A5.

4) 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 splitted 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 [43] on the collected tokens. After completing the text preprocessing 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 shown in Sections III-A5 and III-A5.

5) Text Feature Extraction: We use two text mining techniques to extract text features: 'bag-of-words (BOW)' [20] and 'term frequency-inverse document frequency (TF-IDF)' [21]. The BOW technique which has been extensively used in software engineering [28] [27] [29], 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 [44] [45]. The TF-IDF technique accounts for the relative frequency of tokens that appear in scripts. Along with BOW, we also include TF-IDF, as prior research has demonstrated that TF-IDF can help in building better prediction models [21].

Bag-of-Words: Using the BOW technique, we use the tokens extracted from Section III-A4. We compute the occurrences of tokens for each script. By using the occurrences of tokens we 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>

Fig. 2: A hypothetical example to illustrate the BOW technique discussed in Section III-A5.

We use a hypothetical example shown in Figure 2 to illustrate the BOW technique. In our hypothetical 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 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 [21]. In our experimental setting, the tokens that we apply TF-IDF on, are extracted from IaC scripts, as discussed in Section III-A4. The documents are the IaC scripts from which we extracted the tokens. For a script s, and a token t, we calculate the TF-IDF as following:

- **Calculate TF**: We calculate TF of token t in script s using Equation 1:

$$TF(t,s) = \frac{\text{occurences of token } t \text{ in script } s}{\text{total count of tokens in script } s} \tag{1}$$

- **Calculate IDF**: We calculate IDF of token t using Equation 2:

$$IDF(t) = \log_{10}\left(\frac{\text{total count of scripts in the dataset}}{\text{count of scripts in which token } t \text{ appears at least once}}\right) \tag{2}$$

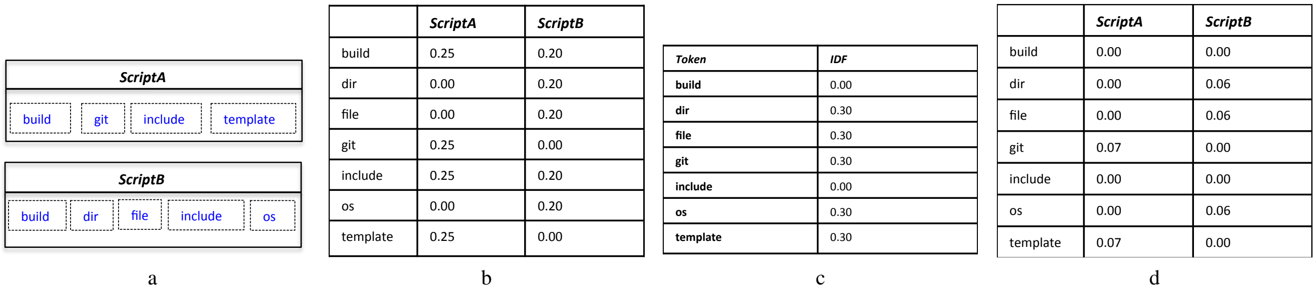
- **Calculate TF-IDF**: We calculate TF-IDF of token t using Equation 3:

$$TF - IDF(t,s) = TF(t,s) * IDF(t) \tag{3}$$

We use a hypothetical example to illustrate how the TF-IDF vectorization process works as shown in Figure 3. In our hypothetical 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 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 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 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.

6) Feature Selection: Feature selection is the process of eliminating features that have minimal influence on prediction performance [46] [22]. For each IaC script we extract text features as tokens using the bag-of-word and TF-IDF techniques. All of the identified tokens might not be correlated with defective IaC scripts and might not contribute in building defect prediction models. The text features that have minimal correlation with defective IaC scripts can be eliminated via feature selection. We use principal component analysis (PCA) [22] for feature selection because PCA accounts for multi-collinearity amongst features [22] and identifies the strongest patterns in the data [22]. 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 text features should be used for building prediction models. We select the count of principal


ScriptA
build
git
include
template
ScriptB
build
dir
file
include
os
a
b
c
d

	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

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

	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

Fig. 3: A hypothetical example to illustrate out feature vectorization technique using TF-IDF. Figure 3a presents the pre-processed tokens of two scripts: $ScriptA$ and $ScriptB$. Figure 3b presents the TF values for both scripts. Figure 3c presents the IDF scores for the unique seven tokens. Finally, TF-IDF scores for each script and token is presented in Figure 3d.

components that account for at least 95% of the total variance to avoid overfitting. The identified components include text features that correlate with defective scripts.

We use the identified components using PCA analysis to answer RQ-1 and RQ-2. As will be described in Section III-B, to answer RQ-1, we apply qualitative analysis on the text features that correlate with defective IaC scripts. Next, as will be described in Section III-C to answer RQ-3, we use the identified principal components as input to statistical learners for building defect prediction models.

B. RQ-1: Characteristics of Defective IaC Scripts

The text features included in the identified principal components from Section III-A6 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) [23]. SGT is a variant of Grounded Theory (GT) [23] [47] that allows for specification of research questions and is used to characterize properties from textual artifacts [23] [47] [48]. SGT includes three elements: 'codes', 'low-level concept', and 'high-level conclusion'. In SGT, a 'high-level conclusion' represents an attribute or property [23], and by deriving these high-level conclusions, we identify properties that characterize defective scripts.

We use an example in Figure 4 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 a high-level conclusion 'Filesystem operations appear more in defective IaC scripts'. This high-level conclusion identifies one of the properties 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 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, 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[2]); 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.

C. RQ-2: Building Prediction Models

We answer RQ-2 in the following manner:

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

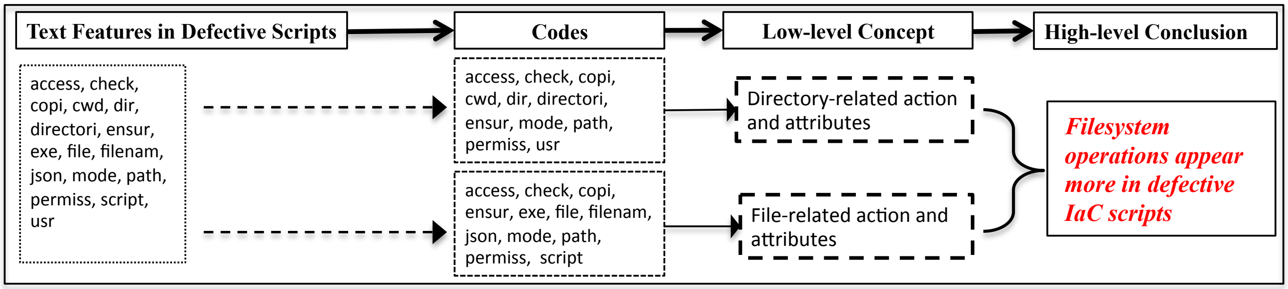

Text Features in Defective Scripts
Codes
Low-level Concept
High-level Conclusion
access, check, copi, cwd, dir, directori, ensur, exe, file, filenam, json, mode, path, permiss, script, usr
access, check, copi, cwd, dir, directori, ensur, mode, path, permiss, usr
access, check, copi, ensur, exe, file,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

Fig. 4: An example of how we use Strauss-Corbin Grounded Theory (SGT) to characterize defective IaC scripts.

1) Statistical Learner: Researchers use statistical learners to build prediction models that learn from historic data and make prediction decisions on unseen data. We use Random Forest (RF), an ensemble technique that creates multiple classification trees, each of which are generated by taking random subsets of the training data [24] [22]. We select RF, as RF does not make any assumptions on the relationship between independent variables and the dependent variable. Researchers [49] recommended the use of statistical learners that uses ensemble techniques to build defect prediction models.

Prediction performance measures: We use two performance metrics to evaluate our prediction models:

- **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 [49], and recommended by prior research [50].
- **F-Measure**: F-Measure is the harmonic mean of precision and recall. Precision calculates the proportion of IaC scripts that are actually defective given that the model predicts as defective. Recall calculates the proportion of defective IaC scripts that are correctly predicted by the prediction model. Increase in precision, often decreases recall, and vice-versa [51] [52]. F-Measure provides a composite score of precision and recall, and is high when both precision and recall is high.

2) Automated Parameter Tuning for Learners: We use differential evolution (DE) [53], a search-based algorithm, to automatically tune the parameters of RF. We select DE because using DE as a parameter tuning technique, researchers have observed improved prediction performance for software defects [35]. 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: $GENERATION$, $POPULATION$, crossover 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. 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. We determine the parameters of RF for tuning from Fu et al.'s paper [35].

3) 10-fold Cross Validation: We use 10-fold cross validation to evaluate the constructed prediction models. In 10-Fold cross validation evaluation approach, the dataset is partitioned into 10 equal sized subsamples or folds [22]. 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 [49], we repeat the 10-fold cross validation 10 times to assess prediction stability.

IV. Case Studies

We use Puppet scripts from open source repositories maintained by three organizations: Mozilla, Openstack, and Wikimedia Commons. We select Puppet because it is considered as one of the most popular tools for configuration management [13] [16], and has been used by IT organizations since 2005 [54]. Mozilla is an open source software community that develops, uses, and supports Mozilla products such as Mozilla Firefox [3]. Openstack foundation is an open-source software platform for cloud computing where virtual servers and other resources are made available to customers [4]. Wikimedia Foundation is a non-profit organization that develops and distributes free educational content [5]. Using the open source repositories we construct three datasets where each Puppet script is labeled as defective or non-defective.

A. Repository Collection

We apply the three selection criteria presented in Section III-A1 to identify the repositories that we use for analysis.

[3] https://www.mozilla.org/en-US/
[4] https://www.openstack.org/
[5] https://wikimediafoundation.org/

We describe how many of the repositories satisfied each of the three criteria as following:

- **Criteria-1**: Altogether, 1594, 1253, and 1638 repositories were publicly available to download respectively, for Mozilla, Openstack, and Wikimedia Commons. We download the repositories respectively from their respective online project management systems (Mozilla [55], Openstack [56], and Wikimedia [57]). The Mozilla repositories were Mercurial-based, whereas, Openstack and Wikimedia repositories were Git-based.
- **Criteria-2**: For Criteria-2, we stated that at least of 11% of all the files belonging to the repository must be Puppet scripts. For Mozilla, 2 of the 1,594 repositories for Mozilla satisfied Criteria-2. For Openstack, 61 of the 1253 repositories satisfied Criteria-2. For Wikimedia Commons, 11 out of the 1,638 repositories satisfied Criteria-2. Altogether 74 of the 4,485 repositories satisfy Criteria-2, indicating the amount of IaC scripts to be small compared to the organizations' overall codebase.
- **Criteria-3**: As Criteria-3, we stated that the repository must have at least two commits per month. The 2, 61, and 11 selected repositories that satisfy Criteria-2 also satisfy Criteria-3.

B. Commit Message Processing

For the Mozilla, Openstack, and Wikimedia repositories, we collect 3074, 7808, and 972 commits, respectively. As shown in Table I, for Mozilla we collect 580 Puppet scripts that map to 3,074 commits from the two repositories. For Openstack, we collect 1,383 Puppet scripts that map to 7,808 commits from the 61 repositories. For Wikimedia, we collect 296 Puppet scripts that map to 972 commits from the 11 repositories. Of the 3074, 7808, and 972 commits, 2764, 2252, and 210 commit messages included unique identifiers that map to issues in their respective issue tracking systems. Using these unique identifiers to issue reports, we construct the COCMs.

C. Determining Defect-related Commits

We categorize COCMs 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 COCMs. The first and second rater, respectively, have a professional experience of three and two years in software engineering. The first and second rater respectively took 37.0 and 51.2 hours to complete the categorization.
  - **Openstack**: Two graduate students, separately, apply qualitative analysis on 7,808 COCMs from Openstack repositories. The first and second rater, respectively, have a professional experience of two and one years in software engineering. The first and second rater completed the categorization of the 7,808 XCMs respectively, in 80.0 and 130.0 hours.
  - **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 IRB protocol (IRB#9521) in recruitment of students and assignment of defect categorization tasks. We randomly distribute the 972 COCMs amongst the students such that each COCM is rated by at least two students. 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.
- **Resolution Phase**:
  - **Mozilla**: Of the 3,074 COCMs, we observe agreement for 2,122 COCMs and disagreement for 952 COCMs, with a Cohen's Kappa score of 0.6. Based on Cohen's Kappa score, the agreement level is 'moderate' [58].
  - **Openstack**: Of the 7,808 COCMs, we observe agreement for 3,188 COCMs, and disagreements for 4,620 COCMs. The Cohen's Kappa score was 0.4. Based on Cohen's Kappa score, the agreement level is 'fair' [58].
  - **Wikimedia**: Of the 972 COCMs, we observe agreement for 557 COCMs, and disagreements for 415 COCMs, with a Cohen's Kappa score of 0.7. Based on Cohen's Kappa score, the agreement level is 'substantial' [58].

  The first author of the paper is the resolver, and resolve disagreements for all three datasets.
- **Practitioner Agreement**: Following our methodology (Section III-A3), we report the agreement level between the raters' and the practitioners' categorization for randomly selected 50 COCMs:
  - **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, which according to Landis and Koch [58] indicates 'almost perfect' agreement.
  - **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, which according to Landis and Koch [58] indicates 'substantial' agreement.
  - **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, which indicates 'almost perfect' agreement [58].

Finally, upon applying qualitative analysis we identify defect-related commits. These defect-related commits list the changed Puppet scripts which we use to identify the defective Puppet scripts. We present the count of defect-related commits, and defective Puppet scripts in Table I. 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, installation issues [6], and new features [7]. In case of Openstack and Wikimedia, respectively, 28.8% and 16.4% of the Puppet-related commits include identifiers that map to issues. The constructed datasets

[6] https://bugzilla.mozilla.org/show_bug.cgi?id=773931

[7] https://bugzilla.mozilla.org/show_bug.cgi?id=868974

TABLE I: Defect Datasets Constructed for the Paper

Properties	Dataset		
	Mozilla (MOZ)	Openstack (OST)	Wikimedia (WIK)
Time Period	Aug,2011-Sep,2016	Mar,2011-Sep,2016	Apr,2005-Sep,2016
Puppet Code Size (LOC)	30,272	122,083	17,439
Puppet Commits	3074 of 60992, 5.0%	7808 of 31460, 24.8%	972 of 14717, 6.6%
Defect-related Commits	558 of 3074, 18.1%	1987 of 7808, 25.4%	298 of 972, 30.6%
Defective Puppet Scripts	259 of 580, 44.6%	810 of 1383, 58.5%	161 of 296, 54.4%

are available online [59].

V. Results

In this section we present our empirical findings.

A. RQ-1: Characteristics of Defective Infrastructure as Code Scripts

We identify 2280, 3542, and 2398 unique text features from all 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 III-B, we use these text features to derive properties that characterize defective IaC scripts using SGT. We identify three properties that characterize defective IaC scripts. These properties are: 'filesystem operations', 'infrastructure provisioning', and 'managing user accounts'. All three properties are derived from text features using BOW and TF-IDF. Each of these properties correspond to an operation executed in an IaC script. We list the identified properties that characterize defective IaC scripts with example code snippets in Table II. 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 II 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. [60] 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 [8] 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 II, 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 [61]. IaC tools, such as Puppet, provide API methods to create and manage users and their credentials in the system. According to some practitioners [62], 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.

B. Frequency of the Identified Characteristics

As described in Section III-B, we apply a two-step process to calculate the frequency of the properties that characterize defective scripts. After executing Step-1, 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.

Finally, after completion of Step 2, we report the frequency of identified properties that characterize defective IaC scripts in Table III. The 'Characteristics' column represents a property, and in the 'Frequency' column we report the frequency of each property. 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. We observe the reduction in the count of scripts for which each property is observed, upon application of Step-2. Our findings hint that the keyword-based matching technique can generate a lot of false positives, and manual inspection can filter out these false positives, as demonstrated by Bosu et al. [63], for detecting vulnerable code changes.

Table III 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, considering all three properties, namely filesystem, infrastructure provisioning, and managing user accounts, then instead of using all 580 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

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

TABLE II: 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 III: Frequency of Identified Characteristics

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

TABLE IV: Performance of Defect Prediction Models

Dataset	AUC		F-Measure	
	BOW	**TF-IDF**	**BOW**	**TF-IDF**
MOZ	0.76	0.75	0.74	0.72
OST	0.59	0.55	0.71	0.74
WIK	0.68	0.56	0.73	0.70

respectively in Openstack and Wikimedia can be prioritized for V&V.

C. RQ-2: Building Defect Prediction Models

We report the median AUC and F-Measure for RF when parameter tuning is applied in Table IV. The 'BOW' and 'TF-IDF' columns, respectively, present the median AUC and F-Measure scores for each dataset when the bag-of-words and TF-IDF techniques were used, respectively. For example, for the Mozilla dataset we observe a median AUC of 0.76 when RF is used, when parameter tuning is applied. Considering median AUC, from Table IV we observe the highest prediction performance is observed for Mozilla (median AUC = 0.76), and the lowest for Openstack (median AUC = 0.59). We also observe that for median AUC, the BOW technique provides better prediction performance. Considering median F-Measure, the BOW technique performs better for Mozilla and Wikimedia, whereas the TF-IDF technique performs better for the Openstack dataset. One possible explanation can be related to the amount of text features: the BOW technique may work better for smaller datasets, with smaller amount of text features, whereas TF-IDF works better for datasets with larger amount of text features.

VI. Discussion

We discuss our findings with possible implications:

Prioritizing V&V Efforts: As shown in Table II and Table III, one property 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 21.7%, 14.5%, and 23.4% of the scripts respectively for 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 V&V 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. [60] 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 [54] [64] [65]. While implementing these steps, practitioners might be introducing defects inadvertently. As shown in Table III 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 [32], 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 V&V. 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 answer to RQ-2 provides evidence 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 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.

Future Research: We investigate two techniques to mine text features. Researchers can investigate if other techniques such as topic modeling [66] and word2vec [67] 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.

VII. Limitation

We discuss the limitations of our paper as following:

Text Mining Techniques: 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%, as shown in Table IV.

Granularity of Prediction: 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.

Bias: The constructed datasets are subject to bias of the raters who categorized the COCMs. We mitigate this bias by letting at least two raters review each COCM. For Openstack and Wikimedia around 50% of the COCMs were resolved by the first author, which makes the final categorization biased.

Datasets: We use three datasets to evaluate our methodology. 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.

VIII. Conclusion

IaC scripts provide practitioners the opportunity to build automated deployment pipelines. Similar to software code, IaC scripts can be defective. We focus on identifying characteristics of defective IaC scripts. 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 observe these three properties appear, respectively, in 31.1%, 34.5%, and 42.9% scripts of the Mozilla, Openstack, and Wikimedia dataset. Next, we build prediction models using statistical learners and parameter tuning of statistical learners. Using the BOW technique, we have observed a median F-Measure of 0.74, 0.71, and 0.73 respectively for Mozilla, Openstack, and Wikimedia. With the TF-IDF technique, we have observed a median F-Measure of 0.72, 0.74, and 0.70 respectively for Mozilla, Openstack, and Wikimedia. We hope our findings will facilitate further research in the area of defect analysis of IaC scripts.

References

[1] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04.

New York, NY, USA: ACM, 2004, pp. 86–96. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007524

[2] E. Shihab, Z. M. Jiang, B. Adams, A. E. Hassan, and R. Bowerman, “Prioritizing the creation of unit tests in legacy software systems,” *Software: Practice and Experience*, vol. 41, no. 10, pp. 1027–1048, 2011. [Online]. Available: http://dx.doi.org/10.1002/spe.1053

[3] F. Elberzhager, S. Kremer, J. Mnch, and D. Assmann, “Guiding testing activities by predicting defect-prone parts using product and inspection metrics,” in *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, Sept 2012, pp. 406–413.

[4] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 9–. [Online]. Available: http://dx.doi.org/10.1109/PROMISE.2007.10

[5] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, “Predicting vulnerable software components via text mining,” *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, Oct 2014.

[6] A. S. Nuez-Varela, H. G. Prez-Gonzalez, F. E. Martnez-Perez, and C. Soubervielle-Montalvo, “Source code metrics: A systematic mapping study,” *Journal of Systems and Software*, vol. 128, no. Supplement C, pp. 164 – 197, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217300663

[7] T. Illes-Seifert and B. Paech, “Exploring the relationship of a files history and its fault-proneness: An empirical method and its application to open source programs,” *Information and Software Technology*, vol. 52, no. 5, pp. 539 – 558, 2010, tAIC-PART 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584909002109

[8] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, “Guided mutation testing for javascript web applications,” *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 429–444, May 2015.

[9] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin, “Synthesizing continuous deployment practices used in software development,” in *Proceedings of the 2015 Agile Conference*, ser. AGILE ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/Agile.2015.12

[10] M. Leppnen, S. Mkinen, M. Pagels, V. P. Eloranta, J. Itkonen, M. V. Mntyl, and T. Mnnist, “The highways and country roads to continuous deployment,” *IEEE Software*, vol. 32, no. 2, pp. 64–72, Mar 2015.

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

[12] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, and L. Williams, “The top 10 adages in continuous deployment,” *IEEE Software*, vol. 34, no. 3, pp. 86–95, May 2017.

[13] Y. Jiang and B. Adams, “Co-evolution of infrastructure and source code: An empirical study,” in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR ’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 45–55. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820527

[14] Wikitech, “Incident documentation20160204-Phabricator,” https://wikitech.wikimedia.orgwikiIncident_documentation/20160204-Phabricator, 2016, [Online; accessed 10-October-2017].

[15] H. Hata, O. Mizuno, and T. Kikuno, “Fault-prone module detection using large-scale text features based on spam filtering,” *Empirical Software Engineering*, vol. 15, no. 2, pp. 147–165, Apr 2010. [Online]. Available: https://doi.org/10.1007/s10664-009-9117-9

[16] R. Shambaugh, A. Weiss, and A. Guha, “Rehearsal: A configuration verification tool for puppet,” *SIGPLAN Not.*, vol. 51, no. 6, pp. 416–430, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/2980983.2908083

[17] M. Voelter, *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. USA: CreateSpace Independent Publishing Platform, 2013.

[18] P. Hudak, “Modular domain specific languages and tools,” in *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, Jun 1998, pp. 134–142.

[19] E. Van Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger, “Attribute grammar-based language extensions for java,” in *Proceedings of the 21st European Conference on Object-Oriented Programming*, ser. ECOOP’07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 575–599. [Online]. Available: http://dl.acm.org/citation.cfm?id=2394758.2394796

[20] Z. S. Harris, “Distributional structure,” *WORD*, vol. 10, no. 2-3, pp. 146–162, 1954.

[21] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[22] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining, (First Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[23] A. Strauss and J. Corbin, *Basics of qualitative research techniques*. Sage publications, 1998.

[24] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1010933404324

[25] T. Sharma, M. Fragkoulis, and D. Spinellis, “Does your configuration code smell?” in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR ’16. New York, NY, USA: ACM, 2016, pp. 189–200. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901761

[26] O. Hanappi, W. Hummer, and S. Dustdar, “Asserting reliable convergence for configuration management scripts,” *SIGPLAN Not.*, vol. 51, no. 10, pp. 328–343, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/3022671.2984000

[27] J. Walden, J. Stuckman, and R. Scandariato, “Predicting vulnerable components: Software metrics vs text mining,” in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, Nov 2014, pp. 23–33.

[28] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, “Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits,” in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15. New York, NY, USA: ACM, 2015, pp. 426–437. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813604

[29] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, “Software vulnerability prediction using text analysis techniques,” in *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, ser. MetriSec ’12. New York, NY, USA: ACM, 2012, pp. 7–10. [Online]. Available: http://doi.acm.org/10.1145/2372225.2372230

[30] O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno, “Spam filter based approach for finding fault-prone software modules,” in *Fourth International Workshop on Mining Software Repositories (MSR’07:ICSE Workshops 2007)*, May 2007, pp. 4–4.

[31] A. Weiss, A. Guha, and Y. Brun, “Tortoise: Interactive system configuration repair,” in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[32] P. Labs, “Puppet Documentation,” https://docs.puppet.com/, 2017, [Online; accessed 10-October-2017].

[33] IEEE, “Ieee standard classification for software anomalies,” *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pp. 1–23, Jan 2010.

[34] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “Automated parameter optimization of classification techniques for defect prediction models,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA: ACM, 2016, pp. 321–332. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884857

[35] W. Fu, T. Menzies, and X. Shen, “Tuning for software analytics: Is it really necessary?” *Information and Software Technology*, vol. 76, pp. 135 – 146, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916300738

[36] A. S. Nuez-Varela, H. G. Prez-Gonzalez, F. E. Martnez-Perez, and C. Soubervielle-Montalvo, “Source code metrics: A systematic mapping study,” *Journal of Systems and Software*, vol. 128, pp. 164 – 197, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217300663

[37] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, “Curating github for engineered software projects,” *Empirical Software Engineering*, pp. 1–35, 2017. [Online]. Available: http://dx.doi.org/10.1007/s10664-017-9512-6

[38] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, “On the ”naturalness” of buggy code,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA: ACM, 2016, pp. 428–439. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884848

[39] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, “Towards building a universal defect prediction model with rank transformed predictors,”

Empirical Softw. Engg., vol. 21, no. 5, pp. 2107–2145, Oct. 2016. [Online]. Available: http://dx.doi.org/10.1007/s10664-015-9396-2

[40] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *2008 16th IEEE International Conference on Program Comprehension*, June 2008, pp. 182–191.

[41] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 392–401.

[42] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960. [Online]. Available: http://dx.doi.org/10.1177/001316446002000104

[43] M. F. Porter, "Readings in information retrieval," K. Sparck Jones and P. Willett, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, ch. An Algorithm for Suffix Stripping, pp. 313–316. [Online]. Available: http://dl.acm.org/citation.cfm?id=275537.275705

[44] A.-C. Florea, J. Anvik, and R. Andonie, *Spark-Based Cluster Implementation of a Bug Report Assignment Recommender System*. Cham: Springer International Publishing, 2017, pp. 31–42. [Online]. Available: https://doi.org10.1007978-3-319-59060-8_4

[45] J. Escobar-Avila, E. Parra, and S. Haiduc, "Text retrieval-based tagging of software engineering video tutorials," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 341–343. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.121

[46] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *J. Mach. Learn. Res.*, vol. 3, pp. 1157–1182, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944968

[47] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: A critical review and guidelines," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 120–131. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884833

[48] E. Alégroth and R. Feldt, "On the long-term use of visual gui testing in industrial practice: a case study," *Empirical Software Engineering*, vol. 22, no. 6, pp. 2937–2971, Dec 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9497-6

[49] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 789–800. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818850

[50] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2008.35

[51] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 432–441. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486846

[52] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'"," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 637–640, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70721

[53] R. Storn and K. Price, "Differential evolution &ndash; a simple and efficient heuristic for global optimization over continuous spaces," *J. of Global Optimization*, vol. 11, no. 4, pp. 341–359, Dec. 1997. [Online]. Available: http://dx.doi.org/10.1023/A:1008202821328

[54] J. T. McCune and Jeffrey, *Pro Puppet*, 1st ed. Apress, 2011. [Online]. Available: https://www.springer.com/gp/book/9781430230571

[55] Mozilla, "Mercurial repositories index," hg.mozilla.org, 2017, [Online; accessed 07-October-2017].

[56] Openstack, "OpenStack git repository browser," http://git.openstack.org/cgit/, 2017, [Online; accessed 03-October-2017].

[57] W. Commons, "Projects: Gerrit Wikimedia," https://gerrit.wikimedia.org/r/#/admin/projects/, 2017, [Online; accessed 05-October-2017].

[58] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977. [Online]. Available: http://www.jstor.org/stable/2529310

[59] A. Rahman, "Dataset for Paper: Characterizing Defective Configuration Code Scripts Used For Continuous Deployment," 12 2017. [Online]. Available: https://doi.org/10.6084/m9.figshare.5729535.v1

[60] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications: An empirical study on software development for the cloud," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 393–403. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786826

[61] E. A. Anderson, "Researching system administration," Ph.D. dissertation, 2002, aAI3063285.

[62] G. Keller, "Incident documentation20160204-Phabricator," https://jumpcloud.comblogwhy-user-management-in-chef-and-puppet-is-a-mistake/, 2012, [Online; accessed 10-October-2017].

[63] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: An empirical study," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 257–268. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635880

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

[65] J. Turnbull, *Pulling Strings with Puppet*, 1st ed. Apress, 2007.

[66] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[67] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems*, ser. NIPS'13. USA: Curran Associates Inc., 2013, pp. 3111–3119. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999792.2999959