Information and Software Technology 112 (2019) 148–163

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Source code properties of defective infrastructure as code scripts

Akond Rahman*, Laurie Williams

North Carolina State University, 890 Oval Drive, Raleigh, NC 27606, USA

ARTICLE INFO

Keywords:
Configuration as code
Continuous deployment
Defect prediction
Devops
Empirical study
Infrastructure as code
Puppet

ABSTRACT

Context: In continuous deployment, software and services are rapidly deployed to end-users using an automated deployment pipeline. Defects in infrastructure as code (IaC) scripts can hinder the reliability of the automated deployment pipeline. We hypothesize that certain properties of IaC source code such as lines of code and hard-coded strings used as configuration values, show correlation with defective IaC scripts.

Objective: The objective of this paper is to help practitioners in increasing the quality of infrastructure as code (IaC) scripts through an empirical study that identifies source code properties of defective IaC scripts.

Methodology: We apply qualitative analysis on defect-related commits mined from open source software repositories to identify source code properties that correlate with defective IaC scripts. Next, we survey practitioners to assess the practitioner's agreement level with the identified properties. We also construct defect prediction models using the identified properties for 2439 scripts collected from four datasets.

Results: 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.

Conclusion: 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.

1. Introduction

Continuous deployment is the process of rapidly deploying software or services automatically to end-users [1]. The practice of infrastructure as code (IaC) scripts is essential to implement an automated deployment pipeline, which facilitates continuous deployment [2]. 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 [2,3]. Commercial IaC tools, such as Ansible[4] and Puppet[5], 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[6].

Similar to software source code, the codebase for IaC scripts in an IT organization can be large, containing hundreds of lines of code [4]. IaC scripts are susceptible to human errors [3] and bad coding practices [5], which can eventually introduce defects in IaC scripts [3,6]. Defects in IaC scripts can have serious consequences for IT organizations who rely on IaC scripts to ensure reliability of the constructed automated deployment pipelines. For example, in January 2017, execution of a defective IaC script erased home directories of ~270 users in cloud instances maintained by Wikimedia Commons [7]. In our paper, we focus on identifying source code properties that correlate 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.

* Corresponding author.
E-mail address: aarahman@ncsu.edu (A. Rahman).

[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://puppet.com/.
[6] https://puppet.com/resources/case-study/ambit-energy.
[7] https://wikitech.wikimedia.org/wiki/Incident_documentation/20170118-Labs.

https://doi.org/10.1016/j.infsof.2019.04.013
Received 20 October 2018; Received in revised form 24 April 2019; Accepted 29 April 2019
Available online 2 May 2019
0950-5849/

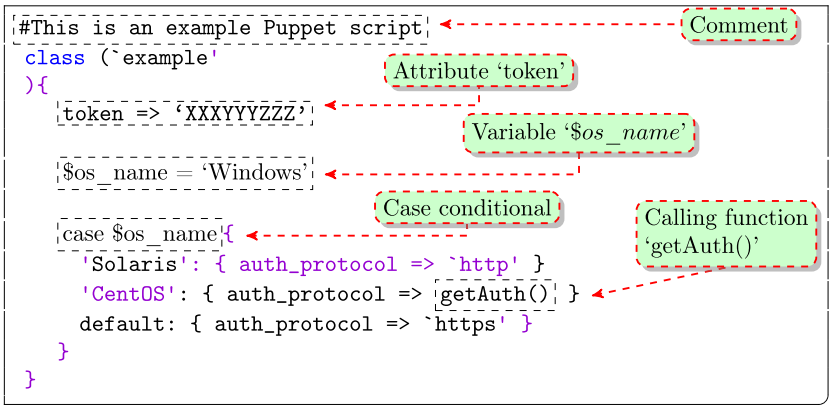

Fig. 1. Annotation of an example Puppet script.

The objective of this paper is to help practitioners in increasing the quality of infrastructure as code (IaC) scripts through an empirical study that identifies source code properties of defective IaC scripts.

We answer the following research questions:

- ***RQ-1**: What source code properties characterize defective infrastructure as code scripts?*
- ***RQ-2**: Do practitioners agree with the identified source code properties?*
- ***RQ-3**: How can we construct defect prediction models for infrastructure as code scripts using the identified source code properties?*

We use 94 open source software (OSS) repositories and collect 12,875 commits that map to 2439 Puppet scripts. Using 89 raters with software engineering experience, we apply qualitative analysis to determine defect-related commits. Using the defect-related commits we determine which of the 2439 scripts are defective. We apply qualitative analysis on defect-related commits to determine which source code properties correlate with defective IaC scripts. We apply statistical analysis to empirically validate the identified properties. We conduct a survey to identify which of the identified properties practitioners agree with. Next, we build defect prediction models using the identified properties and five statistical learners: Classification and Regression Trees [7], K Nearest Neighbor classification [8], Logistic Regression [9], Naive Bayes classification [8], and Random Forest [10] to predict defective IaC scripts. We evaluate the prediction performance of the constructed prediction models using 10×10-fold cross validation [8]. We also compare the prediction performance of our property-based prediction model with prediction approaches built using the (i) bag of words technique, which is used to extract text features from IaC scripts [11], (ii) implementation smells identified by Sharma et al. [12], and (iii) process metrics.

We list our contributions as following:

- A ranked order of source code properties that correlate with defective IaC scripts;
- An evaluation of how practitioners perceive the identified source code properties;
- A set of prediction models built using the identified source code properties to predict defective IaC scripts; and
- A comparison of prediction performance for models constructed using source code properties with text features [4], implementation smells [12] and process metrics.

We organize the rest of the paper as following: we discuss related background and academic work in Section 2. We discuss our methodology, datasets, and results respectively, in Section 3, Section 4, and Section 5. We discuss the implications of our findings in Section 6. We list the limitations of our study in Section 7. Finally, we conclude our paper in Section 8.

2. Background and related work

In this section, we provide background on IaC scripts and briefly describe related academic research.

2.1. Background

IaC is the practice of automatically defining and managing network and system configurations, and infrastructure through source code [2]. Companies widely use commercial tools such as Puppet, to implement the practice of IaC [2,6,13]. We use Puppet scripts to construct our dataset because Puppet is considered one of the most popular tools for configuration management [6,13] and has been used by companies since 2005 [14]. Typical entities of Puppet include modules and manifests [15]. A module is a collection of manifests. Manifests are written as scripts that use a.pp extension.

Puppet provides the utility 'class' that can be used as a placeholder for the specified variables and attributes, which are used to specify configuration values. For attributes, configuration values are specified using the '=>' sign. For variables, configuration values are provided using the '=' sign. Similar to general purpose programming languages, code constructs such as functions/methods, comments, and conditional statements are also available for Puppet scripts. For better understanding, we provide a sample Puppet script with annotations in Fig. 1.

2.2. Related work

Our paper is related to empirical studies that have focused on IaC technologies, such as Puppet. Sharma et al. [12] investigated smells in IaC scripts and proposed 13 implementation and 11 design smells. Hanappi et al. [16] investigated how convergence of Puppet scripts can be automatically tested and proposed an automated model-based test framework. Jiang and Adams [6] investigated the co-evolution of IaC scripts and other software artifacts, such as build files and source code. They reported IaC scripts to experience frequent churn. Weiss et al. [17] proposed and evaluated 'Tortoise', a tool that automatically corrects erroneous configurations in IaC scripts. Hummer at al. [18] proposed a framework to enable automated testing of IaC scripts. Bent et al. [19] proposed and validated nine metrics to detect quality issues in IaC scripts. Rahman et el. [20] investigated which factors influence usage of IaC tools. In another work, Rahman et al. [21] investigated the questions that programmers ask on Stack Overflow to identify the potential challenges programmers face while working with Puppet. Rahman et al. [22] performed a mapping study and identified lack of research studies related to defects in IaC scripts. In another work, Rahman et al. [23] identified seven types of security smells that are indicative of security weaknesses in IaC scripts. They identified 21,201 occurrences of security smells that include 1326 occurrences of hard-coded passwords.

The research study that is closest in spirit was conducted by Rahman and Williams [4]. Rahman and Williams [4] identified certain operations that correlate with defective scripts. Rahman and Williams [4] manually identified if a script includes a certain operation. However, such analysis is coarse-grained: they [4] suggested allocation of inspection efforts for an entire script, which can be effort-intensive. We take a fine-grained approach, where we identify source code properties that show correlation with defective scripts. Our set of properties can help prioritize inspection efforts, for example, instead of inspecting the entire script, practitioners may benefit from focusing on certain properties.

Our paper is also closely related to research studies that have investigated code properties that correlate with defects in source code. Nagappan and Ball [24] 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. [25] investigated how static analysis can be used to identify defects in a large scale industrial software system. They [25] observed that automated static analysis is a relatively affordable fault detection technique compared to that of manual inspection. Zimmermann et al. [26] 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.

The above-mentioned research studies highlight the prevalence of source code properties that correlate with defects in source code. We take motivation from these studies and investigate which source code properties correlate with defective IaC scripts.

3. Methodology

In this section we provide definitions and describe our methodology to answer our research questions.

- **Defect**: An imperfection that needs to be replaced or repaired [27].
- **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.

3.1. *Methodology for dataset construction*

We describe the methodology to construct datasets as following.

3.1.1. *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 [8], Mozilla [9], Openstack [10], and Wikimedia Commons [11]. 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.
- **Criteria-2**: At least 11% of the files belonging to the repository must be IaC scripts. Jiang and Adams [6] 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. [28] used the threshold of at least two commits per month to determine which repositories have enough development activity for software organizations.

[8] https://github.com/Mirantis.
[9] https://hg.mozilla.org/.
[10] https://git.openstack.org/cgit.
[11] https://gerrit.wikimedia.org/r/#/admin/projects/.

For filtering we use our own custom scripts. The repositories collected from Mozilla are Mercurial repositories, whereas the repositories collected from Mirantis, Openstack, and Wikimedia Commons are Git repositories.

3.1.2. *Commit message processing*

Prior research [29–31] 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 [32].
- 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 [32].
- 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 paper. We use the extracted XCMs to separate the defect-related commits from the non-defect-related commits, as described in Section 3.1.3.

3.1.3. *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 an imperfection in an IaC script. We provide raters with a Puppet documentation guide [15] so that raters can obtain background on Puppet. We also provide the raters the IEEE publication on anomaly classification [27] 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 [33] 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 [30]. Defect-related commits list which IaC scripts were changed, and from this list we determine which IaC scripts are defective.

3.2. 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 1021 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 3074 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 7808 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 7808 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 the 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 1021 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' [34].
  - **Mozilla**: Of the 3074 XCMs, we observe agreement for 1308 XCMs and disagreement for 1766 XCMs, with a Cohen's Kappa score of 0.22. Based on Cohen's Kappa score, the agreement level is 'fair' [34].
  - **Openstack**: Of the 7808 XCMs, we observe agreement for 3188 XCMs, and disagreements for 4620 XCMs. The Cohen's Kappa score was 0.21. Based on Cohen's Kappa score, the agreement level is 'fair' [34].
  - **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' [34].

  The first author of the paper 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. [35] 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' [34].
  - **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' [34].
  - **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' [34].
  - **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' [34].

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 first author of the paper 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.

3.3. RQ-1: 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 [36]. We use defect-related XCMs and the code changes performed in defect-related commits that we determined in Section 3.1.3, 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 [32].

Any variant of grounded theory includes three elements: 'concepts', 'categories', and 'propositions' [37]. By deriving propositions, we identify properties and the description behind the identified properties. We use Fig. 3 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 Fig. 3, 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

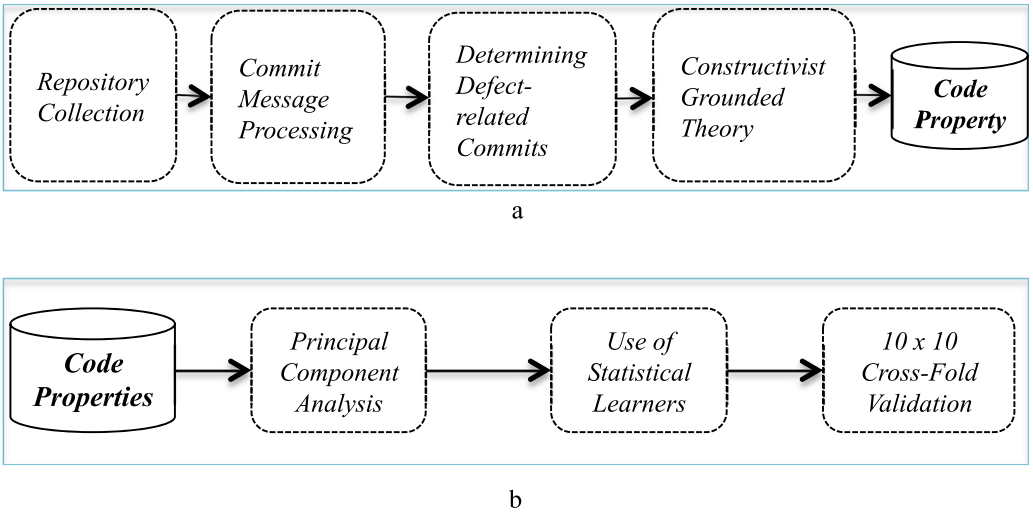

Fig. 2. Methodology: Figs. 2a and b respectively summarizes the methodology for RQ-1 and RQ-3.

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 [12]. We use the Mann-Whitney U test [38] 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 [39] to compare the distribution of each characteristic between defective and neutral files. Both, Mann-Whitney U test and Cliff's Delta 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. [40]'s recommendations to interpret the observed Cliff's Delta values. According to Romano et al. [40], 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 [41]. 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 if a script is defective or neutral. We use Random Forests to build models using all the identified properties as independent variables, and a script of being defective or non-defective, 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. [42]'s recommendations and repeat the process 10 times. We report the median feature importance values for each property, and also apply the Scott–Knott test to statistically determine which property has more feature importance, and thereby exhibits more correlation with defective scripts.

3.4. RQ-2: 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 3.3. Each of the identified properties is presented as a five-point Likert-scale question. Considering the importance of a midpoint in Likert scale items [43], we use a five-point scale: 'Strongly Disagree', 'Disagree', 'Neutral', 'Agree', and 'Strongly Agree'. The survey questions are available online [12].

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

3.5. RQ-3: How can we construct defect prediction models for infrastructure as code scripts using the identified source code properties?

As shown in Fig. 2b, in this section, we provide 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 [44]. As described in Section 3.5.1 we apply principal component analysis (PCA). We use statistical learners to construct defect prediction models, as shown in Section 3.5.2. We evaluate the constructed prediction models using 10 × 10 cross-validation, and four performance measures: AUC, F-Measure, precision, and recall (Section 3.5.3). In Section 3.5.4, we also describe the methods to which we compare our source code properties-based prediction model performance.

3.5.1. Principal component analysis

The identified source code properties using constructivist grounded theory can show implicit correlation with each other, which needs to be accounted for. We use principal component analysis (PCA) [8] to account for multi-collinearity amongst features [8]. PCA has been extensively used in the domain of defect prediction [45,46]. 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. We compute the total amount of variance accounted by the

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

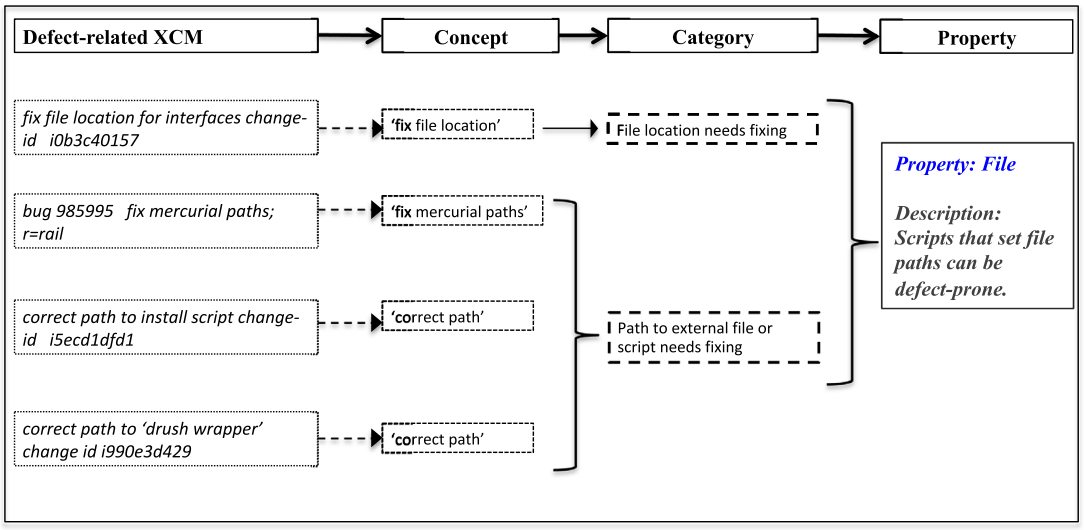

Fig. 3. An example of how we identify source code properties using constructivist grounded theory.

PCA analysis to determine what properties should be used for building prediction models. We select the principal components that account for at least 95% of the total variance to avoid overfitting. Principal components that account for at least 95% of the total variance, are used as input to statistical learners. PCA will not capture the other properties that are not included in our set of identified source code properties.

3.5.2. Statistical learners

Researchers use statistical learners to build prediction models that learn from historic data and make prediction decisions on unseen data. We use the Scikit Learn API [47] to construct prediction models using statistical learners. We use five statistical learners that we briefly describe, and reasons for selecting these learners, as following:

- **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 [7]. We select CART because this learner does not make any assumption on the distribution of features, and is robust to model overfitting [7,8].
- **K Nearest Neighbor (KNN)**: The KNN classification technique stores all available prediction outcomes based on training data and classifies test data based on similarity measures. We select KNN because prior research has reported that defect prediction models that use KNN perform well [48].
- **Logistic Regression (LR)**: LR estimates the probability that a data point belongs to a certain class, given the values of features [9]. LR provides good performance for classification if the features are roughly linear [9]. We select LR because this learner performs well for classification problems [9] such as defect prediction [49] and fault prediction [48].
- **Naive Bayes (NB)**: The NB classification technique computes the posterior probability of each class to make prediction decisions. We select NB because prior research has reported that defect prediction models that use NB perform well [48].
- **Random Forest (RF)**: RF is an ensemble technique that creates multiple classification trees, each of which are generated by taking random subsets of the training data [8,10]. Unlike LR, RF does not expect features to be linear for good classification performance. Researchers [50] recommended the use of statistical learners that uses ensemble techniques to build defect prediction models.

Prediction performance measures: We use four 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 Eq. (1) to calculate precision.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

- **Recall**: Recall measures the proportion of defective IaC scripts that are correctly predicted by the prediction model. We use Eq. (2) to calculate recall.

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

- **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 [50], and recommended by prior research [51].
- **F-Measure**: F-Measure is the harmonic mean of precision and recall. Increase in precision, often decreases recall, and vice-versa [52]. F-Measure provides a composite score of precision and recall, and is high when both precision and recall is high.

Comparing prediction performance: To compare prediction performance using different approaches we use the Scott–Knott test [53]. This variant of Scott–Knott test does not assume input to be normal, and accounts for negligible effect size [53]. The Scott–Knott test uses hierarchical clustering analysis to partition the input data into significantly ($\alpha = 0.05$) distinct ranks [53]. According to the Scott–Knott test, an approach ranks higher if prediction performance of the constructed model using that approach is significantly higher. We use Scott–Knott test to compare for all four prediction performance measures: precision, recall, AUC, and F-measure.

3.5.3. 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 [8]. 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 [50], we repeat the 10-fold cross validation 10 times to avoid prediction errors. We report the median prediction performance score of the 10 runs.

We provide a concrete example to illustrate the whole process. We first apply log transformation on the count of the identified source code properties. Next, we build models using the learners with CART, KNN,

LR, NB, and RF. We evaluate the constructed prediction models using 10 × 10-fold cross validation.

3.5.4. Comparison model construction

For comparison we use three techniques: (i) the text feature-based technique [4], (ii) implementation smells [12]; and (iii) process metrics. Rahman and Williams [4] 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 [54]. We use the 'bag-of-words (BOW)' [11] technique to construct prediction models to compare our identified property-based prediction models. The BOW technique which has been extensively used in software engineering [54], 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.

Text Pre-processing: Before creating text features using bag-of-words, we apply the following text pre-processing 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 [55] 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 shown in Section 3.5.4.

Bag-of-Words (BOW): Using the BOW technique, we use the tokens extracted from text pre-processing step. 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.

We use a hypothetical example shown in Fig. 4 to illustrate the BOW technique. In our hypothetical example, our dataset has two IaC scripts *ScriptX* and *ScriptY* that respectively contain four and five pre-processed tokens. From the occurrences of tokens, we construct a feature matrix where the token 'ci' appears once for *ScriptX* and *ScriptY*.

Implementation smells: We also compare the performance of our prediction models using source code properties with Sharma et al. [12]'s implementation smells. These smells and their corresponding definitions are listed in Table 1.

Process Metrics: We use five commonly used process metrics to compare the performance of our source code-based prediction models. These metrics are:

- *Commits*: Total number of commits made to a script. Similar to prior work, we hypothesize more commits are indicative of defective scripts.

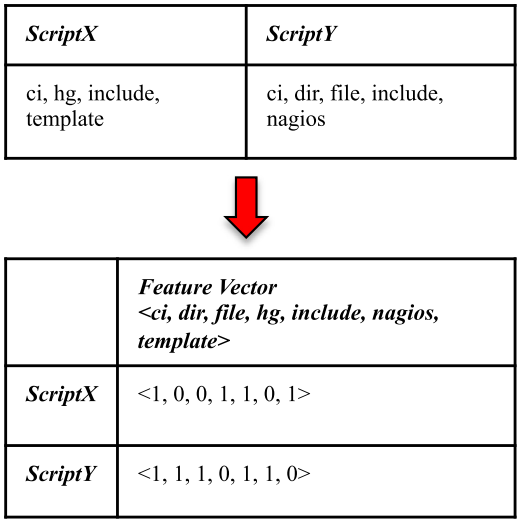

Fig. 4. A hypothetical example to illustrate the BOW technique discussed in Section 3.5.4.

Table 2
Filtering criteria to construct defect datasets.

Criteria	Dataset			
	Mirantis	Mozilla	Openstack	Wikimedia
Criteria-1	26	1594	1253	1638
Criteria-2	20	2	61	11
Criteria-3	20	2	61	11
Final	20	2	61	11

- *Developers*: Total number of developers who made changes to a script. Similar to prior work [56], we hypothesize more developers to be correlated with defective scripts.
- *Fix-related commits*: Total number of commits that are used to fix a defect. We determine fix-related commits if the word 'fix' appears in XCMs, similar to prior work [57].
- *Age*: Age of a script measured in months. Age can be correlated with defective scripts [58].
- *Average Edit Time*: The time difference between edits for a script. The time between edits on a script is correlated with defective scripts [58].

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 Scott–Knott test to compare if the our properties derived using the con-

Table 1
List of implementation smells proposed by Sharma et al. [12].

Smell Name	Description
Missing Default Case	The default case is missing
Inconsistent Naming	Names deviates from convention recommended by configuration tool vendors
Complex Expression	The script contains one or many difficult-to-understand complex expressions
Duplicate Entity	The script contains duplicate hash keys or parameters
Misplaced Attribute	Placement of attributes within a resource or a class does not follow a recommended order
Improper Alignment	The code is not properly aligned
Invalid Property Value	The script contains invalid value of a property or an attribute
Incomplete Tasks	The script includes comments that has 'fixme' and 'todo' as keywords
Deprecated Statement Usage	The script uses one of the deprecated statements
Improper Quote Usage	Single and double quotes are misused in the script
Long Statement	The script contains long statements
Incomplete Conditional	A terminating 'else' clause in an if-else block
Unguarded Variable	A variable is not enclosed in braces when being interpolated in a string

Table 3
Summary statistics of constructed datasets.

Statistic	Dataset			
	Mirantis	Mozilla	Openstack	Wikimedia
Puppet Scripts	180	580	1383	296
Defective Puppet Scripts	91 of 180, 50.5%	259 of 580, 44.6%	810 of 1383, 58.5%	161 of 296, 54.4%
Commits with Puppet Scripts	1021	3074	7808	972
Defect-related Commits	344 of 1021, 33.7%	558 of 3074, 18.1%	1987 of 7808, 25.4%	298 of 972, 30.6%

Table 4
Determining defect categories based on ODC adapted from [59].

Category	Definition	Example XCM
Algorithm (AL)	Indicates efficiency or correctness problems that affect implementation of a feature and can be fixed by re-implementing an algorithm or local data structure	*"fixing deeper hash merging for firewall"*
Assignment (AS)	Indicates defects that are syntax-related, which induces changes in a few lines of code	*"fix missing slash in puppet file url"*
Build/Package/Merge (B)	Indicates defects due to mistakes in change management, library systems, or version control system	*"remove unnecessary package 'requires' that are covered by require_package"*
Checking (C)	Indicates defects related to data validation and value checking	*"bug 1118354: ensure deploystudio user uid is > 500"*
Documentation (D)	Indicates defects that affect maintenance notes	*"fix hadoop.pp documentation default"*
Function (F)	Indicates defects that affect significant capabilities	*"fix for jenkins swarm slave variables"*
Interface (I)	Indicates defects in interacting with other components, modules, or control blocks	*"fix file location for interfaces"*
Other (O)	Indicates a defect that does not belong to the categories: AL, AS, B, C, D, F, I, N, and T	*"fuel-stats nginx fix"*
Timing/Serialization (T)	Indicates errors that involve real time resources and shared resources	*"fix minimal available memory check change-id:iaad0"*

structivist 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.

4. 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 [6,13], and has been used by organizations since 2005 [14]. Mirantis is an organization that focuses on the development and support of cloud services such as OpenStack[13]. Mozilla is an OSS community that develops, uses, and supports Mozilla products such as Mozilla Firefox[14]. Openstack foundation is an OSS platform for cloud computing where virtual servers and other resources are made available to customers[15]. Wikimedia Foundation is a non-profit organization that develops and distributes free educational content[16].

4.1. 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 2. 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.

4.2. Commit message processing

We report summary statistics on the collected repositories in Table 3. According to Table 3, for Mirantis we collect 180 Puppet scripts that map to 1021 commits. The constructed datasets used for empirical analysis are available online[17].

4.3. Defect categorization

We also asked raters to identify defect categories using the defect type attribute of the Orthogonal Defect Classification (ODC) scheme [59]. The ODC scheme includes eight defect categories as shown in Table 4. We added the 'Other' category to include defects that does not belong to any of the eight categories. We report the defect categories and an example XCM for each category of defect type in Table 4.

5. Empirical findings

We report our findings in this section.

5.1. RQ-1: What source code properties characterize defective infrastructure as code scripts?

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 3.1 we identify 12 properties of IaC scripts. Each of these properties are listed in Table 5 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 and Fig. 5. In Table 6 we report the average and maximum value for each property. Fig. 5 presents the box plots for each property. We observe IaC scripts can be as large as 1287 lines of code. Size of IaC scripts further highlights the importance on identifying source code properties: inspecting certain properties within a script could require less effort then inspecting an entire script for an operation.

The median values of 12 source code properties for both defective and non-defective scripts in presented in Table 7. The 'D' and 'ND' respectively presents the median values of each property for defective and

[13] https://www.mirantis.com/.
[14] https://www.mozilla.org/en-US/.
[15] https://www.openstack.org/.
[16] https://wikimediafoundation.org/.
[17] https://figshare.com/s/ad26e370c833e8aa9712.

Table 5
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

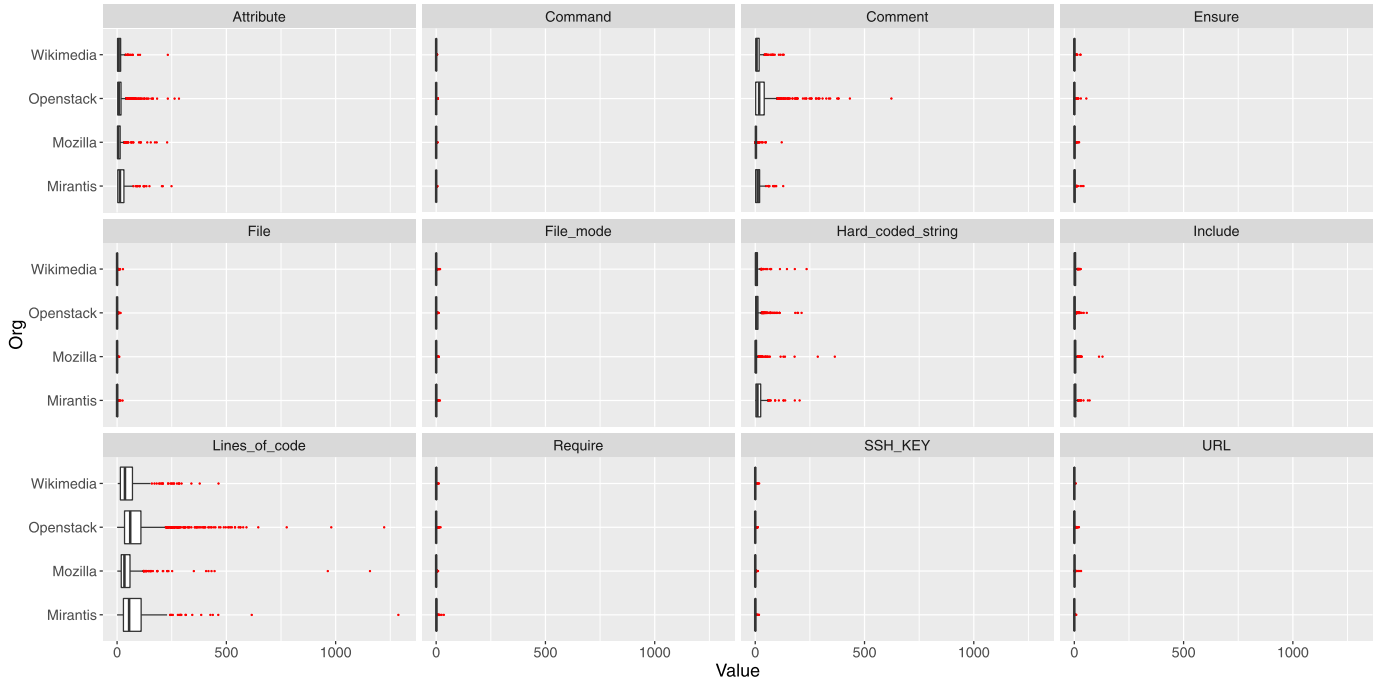

Fig. 5. Distribution of property values for each dataset.

Table 6
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)

Table 7
Median values of 12 source code properties for both defective and non-defective scripts.

Property	Mirantis		Mozilla		Openstack		Wikimedia	
	D	ND	D	ND	D	ND	D	ND
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

non-defective scripts. For example, in the case of the Mirantis, the median values for 'attribute' is respectively 23.0 and 6.5.

In Table 8, 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 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 9. 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'.

Table 8
Validation of identified source code properties. Highlighted cells in bold 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 9
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%)

Table 10
Survey responses from practitioners. Of the 12 properties majority of the practitioners agreed with 'include'.

	Strongly disagree (%)	Disagree (%)	Neutral (%)	Agree (%)	Strongly agree (%)
Attribute	11.5	26.9	34.6	23.0	3.8
Comment	15.3	38.4	23.0	15.3	7.7
Command	3.8	3.8	46.1	30.7	15.3
Ensure	3.8	38.4	38.4	7.7	11.53
File	3.8	15.3	50.0	23.0	7.7
File mode	3.8	19.2	61.5	11.5	3.8
Hard-coded string	3.8	11.5	26.9	46.1	11.5
Include	3.8	11.5	23.0	46.1	15.3
Lines of code	7.7	15.3	34.6	34.6	7.7
Require	3.8	23.0	30.7	34.6	7.7
Ssh_key	3.8	15.3	57.7	15.3	7.7
URL	3.8	11.5	30.7	46.1	3.8

Table 11
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%)

Table 12
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

Our findings related to feature importance is in congruence with our findings presented in Table 8. 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 9, '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 [17] are: 'attribute', 'file mode', 'file', 'hard-coded string', and 'URL'.

Table 13
AUC for each model building technique. The highlighted cell in bold indicates the best technique, as determined by the Scott–Knott test.

Dataset	Property-based					Bag-of-words				
	CART	KNN	LR	NB	RF	CART	KNN	LR	NB	RF
MIR	0.65	0.67	**0.71**	0.62	0.65	0.61	0.65	0.57	0.64	0.66
MOZ	**0.71**	0.66	0.69	0.66	0.69	0.52	0.48	0.51	0.60	0.56
OST	0.52	0.54	0.63	**0.66**	0.54	0.55	0.55	0.64	0.63	0.56
WIK	0.64	0.65	**0.68**	0.64	0.64	0.57	0.52	0.47	**0.68**	0.61

Table 14
Precision for each model building technique. The highlighted cell in bold indicates the best technique, as determined by the Scott–Knot test.

Dataset	Property-based					Bag-of-words				
	CART	KNN	LR	NB	RF	CART	KNN	LR	NB	RF
MIR	0.65	0.69	0.78	**0.80**	0.68	0.62	0.74	0.63	0.75	0.69
MOZ	0.68	0.63	0.73	**0.85**	0.67	0.51	0.41	0.48	0.39	0.58
OST	0.60	0.62	0.70	**0.84**	0.62	0.63	0.64	0.65	0.76	0.64
WIK	0.67	0.68	0.74	**0.85**	0.68	0.60	0.60	0.51	0.76	0.64

Table 15
Recall for each model building technique. The highlighted cell in bold indicates the best technique, as determined by the Scott–Knott test.

Dataset	Property-based					Bag-of-words				
	CART	KNN	LR	NB	RF	CART	KNN	LR	NB	RF
MIR	0.66	**0.70**	0.63	0.34	0.66	0.69	0.49	0.48	0.45	0.64
MOZ	**0.66**	0.61	0.54	0.37	0.64	0.25	0.22	0.21	0.39	0.27
OST	0.60	0.60	**0.67**	0.42	0.58	0.62	0.50	0.57	0.46	0.57
WIK	**0.67**	**0.67**	0.63	0.35	0.63	0.65	0.24	0.30	0.59	0.64

5.2. RQ-2: Do practitioners agree with the identified source code properties?

As mentioned in Section 3.4 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 11. 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 our feature importance analysis. The least agreed property is 'File mode'. Reported agreement level by all practitioners is presented in Table 10. 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 also compare practitioner survey responses and our feature importance analysis by presenting the practitioner agreement level in Table 9. 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, majority of the practitioners agreed with 'include' in contrary to the feature importance analysis for the four datasets.

On the contrary to our statistical analysis, we observe practitioners to show highest agreement with 'include'. One possible explanation can be related to the exposure and use of Puppet. Perhaps, use of Puppet is relatively new for infrastructure automation, and not all practitioners aware of other uses of Puppet.

To gain further insights we invited 20 practitioners for an interview from December 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 doesn't 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[18] 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 reasons for disagreements can be attributed to the experience of the practitioners.

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

5.3. RQ-3: How can we construct defect prediction models for infrastructure as code scripts using the identified source code properties?

As described in Section 3.5, we use PCA analysis to construct prediction models needed for RQ-3. We report the number of principal components that account for at least 95% of the data variability in Table 12.

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

Table 16
F-measure for each model building technique. The highlighted cell in bold indicates the best technique, as determined by the Scott–Knott test.

Dataset	Property-based					Bag-of-words				
	CART	KNN	LR	NB	RF	CART	KNN	LR	NB	RF
MIR	0.67	**0.70**	**0.70**	0.48	0.67	0.66	0.59	0.64	0.63	0.67
MOZ	**0.67**	0.62	0.62	0.52	0.65	0.34	0.29	0.29	0.48	0.37
OST	0.60	0.61	**0.68**	0.56	0.60	0.62	0.56	0.61	0.58	0.60
WIK	**0.67**	**0.67**	0.68	0.50	0.66	0.63	0.35	0.38	0.66	0.65

Table 17
Comparing AUC between our identified source code properties and IaC implementation smells proposed by Sharma et al. [12] for each model building technique. The highlighted cell in bold indicates the best technique, as determined by the Scott–Knott test.

Dataset	Property-based					Implementation Smells [12]				
	CART	KNN	LR	NB	RF	CART	KNN	LR	NB	RF
MIR	0.65	0.67	**0.71**	0.62	0.65	0.48	0.44	0.49	0.49	0.46
MOZ	**0.71**	0.66	0.69	0.66	0.69	0.60	0.57	0.64	0.62	0.59
OST	0.52	0.54	0.63	**0.66**	0.54	0.48	0.48	0.50	0.52	0.49
WIK	0.64	0.65	**0.68**	0.64	0.64	0.47	0.48	0.45	0.50	0.46

Table 18
Comparing precision between our identified source code properties and IaC implementation smells proposed by Sharma et al. [12] for each model building technique. The highlighted cell in bold indicates the best technique, as determined by the Scott–Knott test.

Dataset	Property-based					Implementation Smells [12]				
	CART	KNN	LR	NB	RF	CART	KNN	LR	NB	RF
MIR	0.65	0.69	0.78	**0.80**	0.68	0.52	0.48	0.52	0.51	0.51
MOZ	0.68	0.63	0.73	**0.85**	0.67	0.60	0.55	0.73	0.77	0.58
OST	0.60	0.62	0.70	**0.84**	0.62	0.57	0.57	0.58	0.74	0.57
WIK	0.67	0.68	0.74	**0.85**	0.68	0.52	0.53	0.52	0.61	0.52

Table 19
Comparing recall between our identified source code properties and IaC implementation smells proposed by Sharma et al. [12] for each model building technique. The highlighted cell in bold indicates the best technique, as determined by the Scott–Knott test.

Dataset	Property-based					Implementation Smells [12]				
	CART	KNN	LR	NB	RF	CART	KNN	LR	NB	RF
MIR	0.66	0.70	0.63	0.34	0.66	0.90	0.52	**0.97**	0.23	0.83
MOZ	**0.66**	0.61	0.54	0.37	0.64	0.47	0.48	0.41	0.32	0.48
OST	0.60	0.60	0.67	0.42	0.58	0.84	0.86	0.91	0.09	**0.96**
WIK	0.67	0.67	0.63	0.35	0.63	**0.93**	0.80	0.90	0.05	0.81

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 13. 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 14, 15, and 16. 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 Table 14, for NB we observe the highest median precision for all four datasets, where the median precision is greater than 0.80. According to Table 15, CART provides the highest median recall for two datasets, whereas the highest median recall is obtained for KNN and LR respectively for Mirantis and Openstack. CART, KNN, and LR provides the highest median F-measure for two datasets according to Table 16. For three measures precision, recall, and F-measure, our property-based prediction models outperform the bag-of-words technique.

In Tables 17–19, and 20 we respectively compare the median AUC, precision, recall and F-measure values using our set of source code properties and Sharma et al. [12]'s implementation smells. We observe that our set of source code properties outperforms the implementation smell-based model for AUC and precision. In case of recall, implementation smells perform better than the source code properties for three datasets. For F-measure, implementation smell-based models outperform our source code property-based models for one dataset. Based on our find-

Table 20
Comparing F-measure between our identified source code properties and IaC implementation smells proposed by Sharma et al. [12] for each model building technique. The highlighted cell in bold indicates the best technique, as determined by the Scott–Knott test.

Dataset	Property-based					Implementation Smells [12]				
	CART	KNN	LR	NB	RF	CART	KNN	LR	NB	RF
MIR	0.67	**0.70**	**0.70**	0.48	0.67	0.66	0.50	0.68	0.68	0.31
MOZ	**0.67**	0.62	0.62	0.52	0.65	0.53	0.52	0.53	0.45	0.53
OST	0.60	0.61	0.68	0.56	0.60	0.71	0.69	**0.73**	0.16	0.72
WIK	0.67	0.67	**0.68**	0.50	0.66	0.67	0.64	0.66	0.09	0.63

Table 21
Comparing AUC between our identified source code properties and process metrics for each model building technique. The highlighted cell in bold indicates the best technique, as determined by the Scott–Knott test.

Dataset	Property-based					Process				
	CART	KNN	LR	NB	RF	CART	KNN	LR	NB	RF
MIR	0.65	0.67	0.71	0.62	0.65	0.60	0.63	**0.76**	0.73	0.64
MOZ	0.71	0.66	0.69	0.66	0.69	0.60	0.58	0.63	**0.75**	0.54
OST	0.52	0.54	0.63	0.66	0.54	0.56	0.58	**0.73**	0.63	0.57
WIK	0.64	0.65	**0.68**	0.64	0.64	0.55	0.60	0.67	0.62	0.53

Table 22
Comparing precision between our identified source code properties and process metrics for each model building technique. The highlighted cell in bold indicates the best technique, as determined by the Scott–Knott test.

Dataset	Property-based					Process				
	CART	KNN	LR	NB	RF	CART	KNN	LR	NB	RF
MIR	0.65	0.69	0.78	0.80	0.68	0.63	0.64	0.80	**0.83**	0.66
MOZ	0.68	0.63	0.73	**0.85**	0.67	0.57	0.55	0.71	0.75	0.53
OST	0.60	0.62	0.70	**0.84**	0.62	0.64	0.65	0.78	0.79	0.64
WIK	0.67	0.68	0.74	**0.85**	0.68	0.60	0.64	0.74	0.67	0.57

Table 23
Comparing recall between our identified source code properties and process metrics for each model building technique. The highlighted cell in bold indicates the best technique, as determined by the Scott–Knott test.

Dataset	Property-based					Process				
	CART	KNN	LR	NB	RF	CART	KNN	LR	NB	RF
MIR	0.66	0.70	0.63	0.34	0.66	0.62	**0.73**	0.72	0.59	0.65
MOZ	**0.66**	0.61	0.54	0.37	0.64	0.57	0.52	0.40	0.24	0.49
OST	0.60	0.60	0.67	0.42	0.58	0.60	0.65	**0.72**	0.42	0.65
WIK	**0.67**	**0.67**	0.63	0.35	0.63	0.50	0.58	0.57	0.60	0.48

Table 24
Comparing F-measure between our identified source code properties and process metrics for each model building technique. The highlighted cell in bold indicates the best technique, as determined by the Scott–Knott test.

Dataset	Property-based					Process				
	CART	KNN	LR	NB	RF	CART	KNN	LR	NB	RF
MIR	0.67	0.70	0.70	0.48	0.67	0.62	0.68	**0.76**	0.69	0.66
MOZ	**0.67**	0.62	0.62	0.52	0.65	0.57	0.54	0.51	0.36	0.51
OST	0.60	0.61	0.68	0.56	0.60	0.62	0.65	**0.75**	0.54	0.65
WIK	0.67	0.67	**0.68**	0.50	0.66	0.55	0.61	0.65	0.63	0.52

ings, we suggest that if false positives is not a concern and practitioners want to identify all defective scripts, implementation smells may be a better approach compared to our source code property-based approach.

In Tables 21–23, and 24 we respectively compare the median AUC, precision, recall and F-measure values using our set of source code properties and the process metrics: commits, developers, fix-related commits, age, and average edit time.

For AUC, process metrics perform better than our set of source code properties. With respect to precision, our source code-based prediction models is better than process metrics for three datasets. For two datasets source code properties outperform process metrics based on recall and F-measure values. Based on our findings we recommend to explore both types of metrics: source code-based and process metrics for IaC defect prediction models.

6. Discussion

We discuss our findings with possible implications as following.

6.1. Implications for practitioners

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. As shown in Tables 8 and 9, '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. From Table 9 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 1287 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 [60] 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[19]. 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.

6.2. Future research

Our paper provides opportunity for further research in the area of defect prediction of IaC scripts. Sophisticated statistical techniques, such as topic modeling and deep learning, can be applied to discover more IaC-related source code properties. Researchers can also investigate how practitioners in real life perceive and use defect prediction models for IaC scripts.

7. Threats to validity

We discuss the limitations of our paper as following:

- **Conclusion Validity**: 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 at leeast two 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.

  For RQ-2 the survey response rate was 7.4%. We acknowledge that the survey response rate was low and our findings may not be generalizable.
- **Internal Validity**: We have used a combination of commit messages and issue report descriptions to determine if an IaC script is associated with a defect. We acknowledge that these messages might not have given the full context for the raters. Other sources of information such as practitioner input and code changes that take place in each commit could have provided the raters better context to categorize the XCMs.

  We acknowledge that our set of properties is not comprehensive. 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.

  Multiple raters may miss a defect-related commit, which may impact the distribution of the defect-related commits in our constructed datasets.
- **Construct validity**: 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.
- **External Validity**: Our scripts 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.

  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.

8. 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 of IaC scripts. We apply qualitative analysis 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. Using our 12 properties we construct defect prediction models, which outperform the bag-of-words technique with respect to precision, recall, and F-measure. We hope our paper will facilitate further research in the area of defect analysis for IaC scripts.

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

Conflict of interest

None.

Acknowledgments

We thank the practitioners for responding to our e-mails related to defect categorization and survey analysis. We also thank the members of the RealSearch group for their valuable feedback. Our research paper is partially supported by the National Security Agency (NSA)'s Science of Security Lablet at the North Carolina State University.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.infsof.2019.04.013.

References

[1] A.A.U. Rahman, E. Helms, L. Williams, C. Parnin, Synthesizing continuous deployment practices used in software development, in: Proceedings of the 2015 Agile Conference, in: AGILE '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 1–10, doi:10.1109/Agile.2015.12.

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

[3] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, L. Williams, The top 10 adages in continuous deployment, IEEE Softw. 34 (3) (2017) 86–95, doi:10.1109/MS.2017.86.

[4] A. Rahman, L. Williams, Characterizing defective configuration scripts used for continuous deployment, in: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 2018, pp. 34–45, doi:10.1109/ICST.2018.00014.

[5] J. Cito, P. Leitner, T. Fritz, 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, in: ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 393–403, doi:10.1145/2786805.2786826.

[6] Y. Jiang, B. Adams, Co-evolution of infrastructure and source code: An empirical study, in: Proceedings of the 12th Working Conference on Mining Software Repositories, in: MSR '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 45–55.

[7] L. Breiman, J. Friedman, R.A. Olshen, C.J. Stone, Classification and Regression Trees, 1st, Chapman & Hall, New York, 1984.

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

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

[10] L. Breiman, Random forests, Machine Learning 45 (1) (2001) 5–32, doi:10.1023/A:1010933404324.

[11] Z.S. Harris, Distributional structure, WORD 10 (2–3) (1954) 146–162, doi:10.1080/00437956.1954.11659520.

[12] T. Sharma, M. Fragkoulis, D. Spinellis, Does your configuration code smell? in: Proceedings of the 13th International Conference on Mining Software Repositories, in: MSR '16, ACM, New York, NY, USA, 2016, pp. 189–200, doi:10.1145/2901739.2901761.

[13] R. Shambaugh, A. Weiss, A. Guha, Rehearsal: a configuration verification tool for puppet, SIGPLAN Not. 51 (6) (2016) 416–430, doi:10.1145/2980983.2908083.

[14] J.T. McCune, Jeffrey, Pro Puppet, Apress, 2011, doi:10.1007/978-1-4302-3058-8. 1 edition.

[15] P. Labs, Puppet Documentation, 2017, (https://docs.puppet.com/).[Online; accessed 10-October-2017].

[16] O. Hanappi, W. Hummer, S. Dustdar, Asserting reliable convergence for configuration management scripts, SIGPLAN Not. 51 (10) (2016) 328–343, doi:10.1145/3022671.2984000.

[17] A. Weiss, A. Guha, Y. Brun, Tortoise: Interactive system configuration repair, in: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, in: ASE 2017, IEEE Press, Piscataway, NJ, USA, 2017, pp. 625–636.

[18] W. Hummer, F. Rosenberg, F. Oliveira, T. Eilam, Automated testing of chef automation scripts, in: Proceedings Demo:38; Poster Track of ACM/IFIP/USENIX International Middleware Conference, in: MiddlewareDPT '13, ACM, New York, NY, USA, 2013, pp. 4:1–4:2, doi:10.1145/2541614.2541632.

[19] E. van der Bent, J. Hage, J. Visser, G. Gousios, How good is your puppet? an empirically defined and validated quality model for puppet, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 164–174, doi:10.1109/SANER.2018.8330206.

[20] A. Rahman, A. Partho, D. Meder, L. Williams, Which factors influence practitioners' usage of build automation tools? in: Proceedings of the 3rd International Workshop on Rapid Continuous Software Engineering, in: RCoSE '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 20–26, doi:10.1109/RCoSE.2017..8.

[21] A. Rahman, A. Partho, P. Morrison, L. Williams, What questions do programmers ask about configuration as code? in: Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering, in: RCoSE '18, ACM, New York, NY, USA, 2018, pp. 16–22, doi:10.1145/3194760.3194769.

[22] A. Rahman, R. Mahdavi-Hezaveh, L. Williams, A systematic mapping study of infrastructure as code research, Inf. Softw. Technol. (2018), doi:10.1016/j.infsof.2018.12.004.

[23] A. Rahman, C. Parnin, L. Williams, The seven sins: Security smells in infrastructure as code scripts, in: Proceedings of the 41st International Conference on Software Engineering, in: ICSE '19, 2019. To appear. Pre-print: https://akondrahman.github.io/papers/ist18_iac_sms.pdf.

[24] N. Nagappan, T. Ball, Use of relative code churn measures to predict system defect density, in: Proceedings of the 27th International Conference on Software Engineering, in: ICSE '05, ACM, New York, NY, USA, 2005, pp. 284–292, doi:10.1145/1062455.1062514.

[25] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl, M.A. Vouk, On the value of static analysis for fault detection in software, IEEE Trans. Softw. Eng. 32 (4) (2006) 240–253, doi:10.1109/TSE.2006.38.

[26] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, in: PROMISE '07, IEEE Computer Society, Washington, DC, USA, 2007, p. 9, doi:10.1109/PROMISE.2007.10.

[27] Ieee standard classification for software anomalies, IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993) (2010) 1–23, doi:10.1109/IEEESTD.2010.5399061. IEEE.

[28] N. Munaiah, S. Kroh, C. Cabrey, M. Nagappan, Curating github for engineered software projects, Empirical Softw. Eng. (2017) 1–35, doi:10.1007/s10664-017-9512-6.

[29] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, P. Devanbu, On the "naturalness" of buggy code, in: Proceedings of the 38th International Conference on Software Engineering, in: ICSE '16, ACM, New York, NY, USA, 2016, pp. 428–439, doi:10.1145/2884781.2884848.

[30] F. Zhang, A. Mockus, I. Keivanloo, Y. Zou, Towards building a universal defect prediction model with rank transformed predictors, Empirical Softw. Eng. 21 (5) (2016) 2107–2145, doi:10.1007/s10664-015-9396-2.

[31] F. Zhang, A.E. Hassan, S. McIntosh, Y. Zou, The use of summation to aggregate software metrics hinders the performance of defect prediction models, IEEE Trans. Softw. Eng. 43 (5) (2017) 476–491, doi:10.1109/TSE.2016.2599161.

[32] A. Alali, H. Kagdi, J.I. Maletic, What's a typical commit? a characterization of open source software repositories, in: 2008 16th IEEE International Conference on Program Comprehension, 2008, pp. 182–191, doi:10.1109/ICPC.2008.24.

[33] J. Cohen, A coefficient of agreement for nominal scales, Edu. Psychol. Meas. 20 (1) (1960) 37–46.

[34] J.R. Landis, G.G. Koch, The measurement of observer agreement for categorical data, Biometrics 33 (1) (1977) 159–174.

[35] K. Henningsson, C. Wohlin, Assuring fault classification agreement "an empirical evaluation, in: Proceedings of the 2004 International Symposium on Empirical Software Engineering, in: ISESE '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 95–104, doi:10.1109/ISESE.2004.13.

[36] K. Charmaz, Constructing Grounded Theory, Sage Publishing, London, UK, 2014.

[37] N.R. Pandit, The creation of theory: a recent application of the grounded theory method, Qualit. Rep. 2 (4) (1996) 1–20.

[38] H.B. Mann, D.R. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Ann. Math. Stat. 18 (1) (1947) 50–60.

[39] N. Cliff, Dominance statistics: ordinal analyses to answer ordinal questions., Psychol. Bull. 114 (3) (1993) 494–509.

[40] J. Romano, J. Kromrey, J. Coraggio, J. Skowronek, 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? in: annual meeting of the Florida Association of Institutional Research, 2006, pp. 1–3.

[41] D.R. Cutler, T.C. Edwards, K.H. Beard, A. Cutler, K.T. Hess, J. Gibson, J.J. Lawler, Random forests for classification in ecology, Ecology 88 (11) (2007) 2783–2792, doi:10.1890/07-0539.1.

[42] R. Genuer, J.-M. Poggi, C. Tuleau-Malot, Variable selection using random forests, Pattern Recognit. Lett. 31 (14) (2010) 2225–2236, doi:10.1016/j.patrec.2010.03.014.

[43] R. Garland, The mid-point on a rating scale: is it desirable, Marketing Bull. (1991) 66–70.

[44] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Trans. Softw. Eng. 33 (1) (2007) 2–13, doi:10.1109/TSE.2007.256941.

[45] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in: Proceedings of the 28th International Conference on Software Engineering, in: ICSE '06, ACM, New York, NY, USA, 2006, pp. 452–461, doi:10.1145/1134285.1134349.

[46] B. Ghotra, S. Mcintosh, A.E. Hassan, A large-scale study of the impact of feature selection techniques on defect classification models, in: Proceedings of the 14th International Conference on Mining Software Repositories, in: MSR '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 146–157, doi:10.1109/MSR.2017.18.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.

[48] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Trans. Softw. Eng. 38 (6) (2012) 1276–1304, doi:10.1109/TSE.2011.103.

[49] F. Rahman, P. Devanbu, How, and why, process metrics are better, in: Proceedings of the 2013 International Conference on Software Engineering, in: ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 432–441.
[50] B. Ghotra, S. McIntosh, 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, in: ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 789–800.
[51] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: a proposed framework and novel findings, IEEE Trans. Softw. Eng. 34 (4) (2008) 485–496, doi:10.1109/TSE.2008.35.
[52] T. Menzies, A. Dekhtyar, J. Distefano, J. Greenwald, Problems with precision: a response to "comments on 'data mining static code attributes to learn defect predictors'", IEEE Trans. Softw. Eng. 33 (9) (2007) 637–640, doi:10.1109/TSE.2007.70721.
[53] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, K. Matsumoto, An empirical comparison of model validation techniques for defect prediction models, IEEE Trans. Softw. Eng. 43 (1) (2017) 1–18, doi:10.1109/TSE.2016.2584050.
[54] J. Walden, J. Stuckman, R. Scandariato, Predicting vulnerable components: Software metrics vs text mining, in: 2014 IEEE 25th International Symposium on Software Reliability Engineering, 2014, pp. 23–33, doi:10.1109/ISSRE.2014.32.
[55] M.F. Porter, Readings in information retrieval, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997, pp. 313–316.
[56] A. Meneely, L. Williams, Secure open source collaboration: An empirical study of linus' law, in: Proceedings of the 16th ACM Conference on Computer and Communications Security, in: CCS '09, ACM, New York, NY, USA, 2009, pp. 453–462, doi:10.1145/1653662.1653717.
[57] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, N. Ubayashi, An empirical study of just-in-time defect prediction using cross-project models, in: Proceedings of the 11th Working Conference on Mining Software Repositories, in: MSR 2014, ACM, New York, NY, USA, 2014, pp. 172–181, doi:10.1145/2597073.2597075.
[58] A. Tosun Misirli, B. Murphy, T. Zimmermann, A. Basar Bener, An explanatory analysis on eclipse beta-release bugs through in-process metrics, in: Proceedings of the 8th International Workshop on Software Quality, in: WoSQ '11, ACM, New York, NY, USA, 2011, pp. 26–33, doi:10.1145/2024587.2024595.
[59] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, M.Y. Wong, Orthogonal defect classification-a concept for in-process measurements, IEEE Trans. Softw. Eng. 18 (11) (1992) 943–956, doi:10.1109/32.177364.
[60] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, E.J. Whitehead Jr., Does bug prediction support human developers? findings from a Google case study, in: Proceedings of the 2013 International Conference on Software Engineering, in: ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 372–381.