Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts

Akond Rahman
Tennessee Tech University
Tennessee, USA
arahman@tntech.edu

Effat Farhana
NC State University
North Carolina, USA
efarhan@ncsu.edu

Chris Parnin
NC State University
North Carolina, USA
cjparnin@ncsu.edu

Laurie Williams
NC State University
North Carolina, USA
lawilli3@ncsu.edu

ABSTRACT

Defects in infrastructure as code (IaC) scripts can have serious consequences, for example, creating large-scale system outages. A taxonomy of IaC defects can be useful for understanding the nature of defects, and identifying activities needed to fix and prevent defects in IaC scripts. *The goal of this paper is to help practitioners improve the quality of infrastructure as code (IaC) scripts by developing a defect taxonomy for IaC scripts through qualitative analysis.* We develop a taxonomy of IaC defects by applying qualitative analysis on 1,448 defect-related commits collected from open source software (OSS) repositories of the Openstack organization. We conduct a survey with 66 practitioners to assess if they agree with the identified defect categories included in our taxonomy. We quantify the frequency of identified defect categories by analyzing 80,425 commits collected from 291 OSS repositories spanning across 2005 to 2019.

Our defect taxonomy for IaC consists of eight categories, including a category specific to IaC called idempotency (i.e., defects that lead to incorrect system provisioning when the same IaC script is executed multiple times). We observe the surveyed 66 practitioners to agree most with idempotency. The most frequent defect category is configuration data i.e., providing erroneous configuration data in IaC scripts. Our taxonomy and the quantified frequency of the defect categories may help in advancing the science of IaC script quality.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**;

KEYWORDS

bug, category, configuration as code, configuration scripts, defect, devops, infrastructure as code, puppet, software quality, taxonomy

ACM Reference format:
Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. 2020. *Gang of Eight*: A Defect Taxonomy for Infrastructure as Code Scripts. In *Proceedings of 42nd International Conference on Software Engineering, Seoul, Republic of Korea, May 23–29, 2020 (ICSE '20),* 13 pages.
https://doi.org/10.1145/3377811.3380409

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380409

1 INTRODUCTION

Infrastructure as code (IaC) is the practice of automatically maintaining system configurations and provisioning deployment environments using source code [26]. IaC scripts are also known as configuration as code scripts [59], or configuration scripts [68]. Information technology (IT) organizations use tools, such as Chef [1] and Puppet [2] to implement the practice of IaC. The use of IaC has yielded benefits for IT organizations, such as General Motors (GM) and the European Organization for Nuclear Research (CERN). Using Chef, GM increased software deployment frequency by a factor of 21 [20]. CERN uses Puppet to manage 15,000 servers and to process 2,000 terabytes of data everyday [7]. Puppet has helped CERN to minimize service disruptions and reduce deployment time [2].

Despite the above-mentioned benefits, defects with serious consequences can appear in IaC scripts. For example, a defect in an IaC script created an outage resulting in business losses worth of 150 million USD for Amazon Web Services [23]. Other example consequences of defects in IaC scripts include outage for GitHub [19] and deletion of user directories for ~270 users in cloud instances maintained by Wikimedia Commons [12].

A defect taxonomy for IaC scripts can help practitioners understand the nature of defects, and identify possible development activities for defect mitigation. Figure 1 presents an example of a security defect, which exposes users' passwords in logs [3]. The defect is mitigated by adding 'secret=>true', which prevents the password getting exposed in logs [3]. Identification of certain defect categories, such as security defects similar to Figure 1, can help practitioners to make informed decisions on what development activities could be adopted to improve the quality of IaC scripts.

The importance of defect categorization has been recognized by the research community [6, 39, 83, 86]. For example, Linraes-Vásquez et al. [39] stated categorizing vulnerabilities can help Android developers "*in focusing their verification & validation (V&V) activities*". According to Catolino et al. [6], "*understanding the bug type represents the first and most time-consuming step to perform in the process of bug triage, since it requires an in-depth analysis*". A defect taxonomy for IaC, an area that remains unexplored, can help practitioners in understanding the nature of defects, and help in improving IaC script quality through activities such as triaging defects, prioritizing V&V efforts, and measuring IaC script quality.

We observe practitioners asking about defect categories for IaC scripts in online forums such as Reddit:"*I want to adopt Puppet in my organization. Before adoption I want to be aware of the quality issues that may arise in Puppet scripts. Can someone give me some pointers*

[1] https://www.chef.io/
[2] https://puppet.com/
[3] https://bugs.launchpad.net/puppet-ceilometer/+bug/1328448

```
glance_cache_config{
 'DEFAULT/auth_url' :value=>$auth_url;
 'DEFAULT/admin_tenant_name' :value=>$keystone_tenant;
 'DEFAULT/admin_user' :value=>$keystone_user;
-'DEFAULT/admin_password' :value=>$keystone_password;
+'DEFAULT/admin_password' :value=>$keystone_password, secret=>true;
}
```

Figure 1: Example of a security-related defect where a password is exposed in logs. The defect is mitigated by adding 'secret=>true'.

on what type of bugs/defects appear for Puppet scripts? " [16]. While the forum members offered suggestions on possible categories of defects, e.g., syntax-related defects, these suggestions lack substantiation. We hypothesize that through systematic empirical analysis we can develop a taxonomy, and derive defect categories for IaC using open source software (OSS) repositories.

The goal of this paper is to help practitioners improve the quality of infrastructure as code (IaC) scripts by developing a defect taxonomy for IaC scripts through qualitative analysis.

We answer the following research questions:

- **[Categorization] RQ$_1$**: *What categories of defects appear in infrastructure as code scripts?*
- **[Perception] RQ$_2$**: *How do practitioners perceive the identified defect categories for infrastructure as code scripts?*
- **[Frequency] RQ$_3$**: *How frequently do the identified defect categories appear in infrastructure as code scripts?*

We derive a defect taxonomy for IaC by applying descriptive coding [65] on 1,448 defect-related commit messages collected from OSS repositories maintained by the Openstack organization [44]. We survey 66 practitioners to assess how practitioners perceive the identified defect categories. To automatically identify IaC defect categories, we develop a tool called **A**utomated **C**ategorizer of **I**nfrastructure as code **D**efects (**ACID**), and apply ACID on 80,415 commits collected from 291 OSS repositories hosted by GitHub, Mozilla [42], Openstack [45], and Wikimedia Commons [13].

We list our contributions as following:

- A taxonomy that includes eight defect categories for IaC scripts (Section 3.2);
- An evaluation of how practitioners perceive the identified defect categories (Section 3.3.2);
- An analysis of how frequently the identified defect categories occur (Section 5.3.2); and
- A tool called ACID that automatically identifies instances of defect categories from repositories with IaC scripts (Section 4).

We organize rest of the paper as following: we discuss background and related work in Section 2. We describe the defect categories and survey results in Section 3. We describe the construction and evaluation of ACID in Section 4. We describe our empirical study in Section 5. We discuss our findings and limitations respectively, in Sections 6 and 7. We conclude the paper in Section 8.

2 BACKGROUND AND RELATED WORK

Here, we provide necessary background and discuss related work.

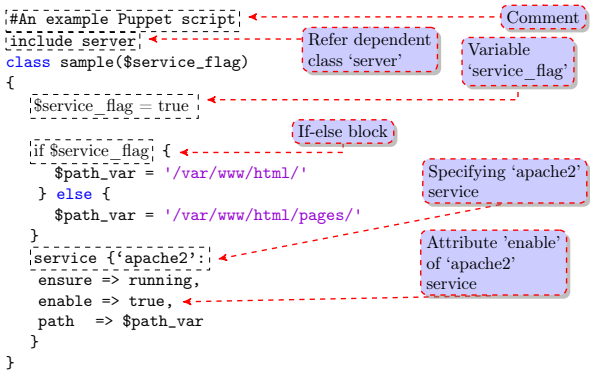

Figure 2: Annotation of an example Puppet script.

2.1 Background

Puppet scripts are analyzed in our research study. Typical entities of Puppet include manifests [36]. Manifests are written as scripts that use a .pp extension. In a single manifest script, configuration data can be specified using variables and attributes. Programming constructs, such as the 'service' resource, are also available to specify services. We provide a sample Puppet script with annotations in Figure 2. In Figure 2, a service with the title 'apache2' is specified. Configuration data are specified using '=>' and '=' respectively, for attributes and variables.

2.2 Related Work

Our paper is closely related to prior research on software defect categories. Chillarege et al. [9] proposed eight defect categories: algorithm, assignment, build, checking, documentation, function, interface, and timing. Categories proposed by Chillarege et al. [9] were used by Cinque et al. [10] to categorize defects for air traffic control software. Wan et al. [80] investigated defects in blockchain systems and observed that semantic defects are the most dominant runtime defect categories. Linares-Vasquez et al. [39] provided a taxonomy to categorize vulnerabilities for Android-based systems. Zheng et al. [84] studied 579 defects collected from the Openstack cloud management system, and observed that 66.1% of the studied defects involve incorrect output. Islam et al. [30] studied 2,716 Stack Overflow posts related to deep learning libraries and observed configuration data to be the most frequent category.

The above-mentioned discussion shows that defect categorization of specialized software such as Android and deep learning provide value to the research community. However, similar research efforts are absent for IaC. Sharma et al. [68] and Bent et al. [78] investigated code maintainability aspects of Chef and Puppet scripts. Jiang and Adams [31] investigated the co-evolution of IaC scripts and other software artifacts, such as build files and source code. Rahman and Williams in separate studies characterized defective IaC scripts using text mining [60], and by identifying source code properties [61]. Hanappi et al. [21] proposed a model-based test framework to automatically test convergence of IaC scripts. Rahman et al. [58] identified 21,201 occurrences of security smells i.e., coding patterns indicative of security weaknesses that also included 1,326 occurrences of hard-coded passwords. The above-mentioned

discussion highlights the lack of studies that investigate defect categories for IaC and motivate us further to investigate defect categories of IaC.

3 DEFECT TAXONOMY FOR INFRASTRUCTURE AS CODE SCRIPTS

In this section, we answer RQ_1: ***What categories of defects appear in infrastructure as code scripts?***. First, we describe our methodology, then we present the identified defect categories.

3.1 Methodology to Develop Defect Taxonomy

Qualitative Analysis: We use descriptive coding [65] on 1,448 defect-related commits. Defect-related commits have commit messages that indicate an action was taken related to a defect [60]. We derive defect-related commits by manually inspecting 7,808 commits that map to 1,386 Puppet scripts. While determining defect-related commits a rater inspected if the commit message expressed an action that was taken to remove or repair an error while developing an IaC script.

We collect the set of 7,808 commits from 61 OSS repositories maintained by the Openstack organization [45], as Openstack provides cloud-based infrastructure services, and our assumption is that an analysis of commits collected from Openstack could give us insights on defect categories. We use commits because commits summarize changes that are made to a script and could identify the types of changes that are being performed on a script. Descriptive coding is a qualitative analysis technique that summarizes the underlying theme from unstructured text [65]. We select descriptive coding because we can obtain (i) a summarized overview of the defect categories that occur for IaC scripts; and (ii) context on how the identified defect categories can be automatically identified. We use Puppet scripts to construct our dataset because Puppet is considered as one of the most popular tools for configuration management [31] [67], and has been used by companies since 2005 [40].

We extract commit message text from the 1,448 defect-related commits, as well as any existing identifier to a bug report in the commit message. We combine the commit message with any existing bug report description and refer to the combination as enhanced commit message (ECM). If no bug identifier is present in the commit message, then the commit message becomes the ECM. We use ECMs to derive defect categories as following: first we identify text patterns that describe a reason and/or a symptom of a defect, where defect is defined as—an imperfection that needs to be replaced or repaired [28]. Figure 3 provides an example of our qualitative analysis process. We first analyze the ECM for each commit where an IaC script is modified, and extract snippets that correspond to a reason or symptom of a defect. From the snippet provided in the bottom left corner, we extract raw text: 'fix config options'. Next, we generate the initial category e.g., we generate 'fixing config. options' from this raw text. Finally, we determine the defect category 'Configuration Data' by combining initial categories. We combine these two initial categories, as both correspond to a common pattern of fixing configuration data defects. Multiple defect categories can be identified in an ECM.

The first and second author, individually, conduct a descriptive coding process on 1,448 defect-related ECMs. Upon completion of this process, we record the agreements and disagreements for the identified defect categories. The first and second author, respectively, identified eight and ten categories. The Cohen's Kappa [11] is 0.8, which according to Landis and Koch [37] is 'substantial agreement'. The second author identified two additional categories: hard-coded values and network setting. Upon discussion, the first and second authors agreed that both hard-coded values and network setting can be merged with the category 'configuration data', as defects related to configuration data include both.

3.2 Answer to RQ_1: Defect Taxonomy for IaC

Our developed taxonomy includes eight defect categories. We observe one category, idempotency, to be unique to IaC, whereas, the other defect categories have been observed for other software systems as reported in prior literature [6, 30, 80, 84]. We report each defect category in an alphabetical order, with examples below:

Conditional: This category represents defects that occur due to erroneous logic and/or conditional values used to execute one or multiple branches of an IaC script. Conditional logic defects also appear for other types of software systems such as, machine learning [76], relational databases [15], [51], text editor software [15], and deep learning software [30].

Example: Conditional logic defects can lead to erroneous status output. For example, for a Wikimedia Commons project 'cdh' [82], when checking the status of mysqladmin, a conditional logic defect caused the output to be '0 (zero)', both in the case of success and failure of mysqladmin's 'ping' command.

Configuration Data: This category represents defects that happen due to erroneous configuration data that reside in IaC scripts. An example configuration data defect downloaded from an OSS repository [46] is *'$config_dir="/etc/$service_name"'*, where wrong configuration data is provided. The fix is *'$config_dir="/etc/hekad"'*. Practitioners have reported configuration data defects in IaC scripts to cause deployment problems for the Google App Engine [52], and inavailability of StackExchange [72], an online question and answer platform [71]. 'Configuration data' includes five sub-categories: (i) data for storage system such as MySQL, MongoDB, and SQLite; (ii) data for file system such as specifying file permissions and file names; (iii) data for network setup such as TCP/DHCP ports and addresses, MAC addresses, and IP table rules; (iv) data for user credentials such as usernames; and (v) data for caching systems such as Memcached.

Prior research has observed configuration data defects to appear for machine learning software [76], build systems [83], blockchain software [80], Eclipse software projects [6], and Mozilla projects [6].

Example: Configuration data-related defects cause deployment failures: in the case of Openstack Bug#1592842 [4], value of an attribute 'host_ip' was not set to the correct IP address, which lead to a deployment task to fail. Other example consequences of configuration data defects include provisioning errors for object storage systems such as Openstack Swift [5].

Dependency: This category represents defects that occur when execution of an IaC script is dependent upon an artifact, which is either missing or incorrectly specified. The artifact can be a file,

[4] https://bugs.launchpad.net/fuel/+bug/1592842
[5] https://bugs.launchpad.net/tripleo/+bug/1532352

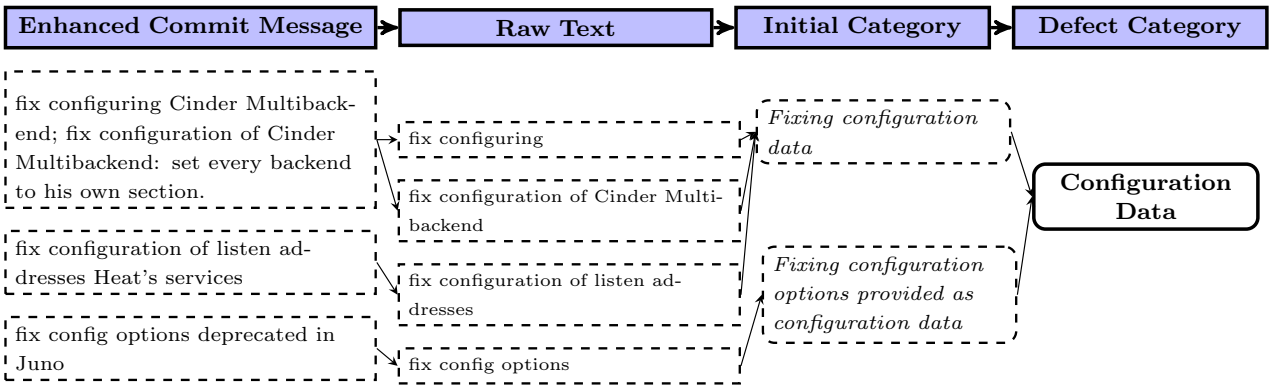

Figure 3: An example to demonstrate the process of deriving IaC defect categories using descriptive coding.

class, package, Puppet manifest, or a module. Previously, practitioners have reported that managing dependencies in IaC scripts leads to complexity, stating dependency management as a 'nightmare' [22], because incorrectly specified dependencies could lead to execution of IaC scripts in the wrong order. Dependency-related defects have been reported for machine learning [76], and audio processing software [15].

Example: Dependency defects cause deployment errors [6]. According to the bug report[6], a user faces an error when installing an Openstack project called 'Neutron' [49], which provides networking as a service in Debian and Ubuntu. The error occurred because of specifying an incorrect package as a dependency. Dependency defects can also cause installation errors [56].

Documentation: This category represents defects that occur when incorrect information about IaC scripts are specified in source code comments, in maintenance notes, and in documentation files such as README files. Documentation-related defects have been reported in prior studies: Chillarege et al. [9] identified documentation as one of the eight defect categories for ODC. Storey et al. [73] and Tan et al. [75] have reported that erroneous information expressed in source comments can lead to defects in source code.

Example: According to a bug report [7], the format of specifying policies for the Openstack Nova project is incorrectly specified in the comments of an IaC script, which can lead to Puppet runtime errors. Other examples of documentation-related defects are providing incompatible license information in comments, and missing license headers [56], which can have negative implications on the distribution of software [79].

Idempotency: This category represents defects that violate the idempotency property for IaC scripts. For IaC, idempotency is the property which ensures that even after n executions, where $n > 1$, the provisioned system's environment is exactly the same as it was after the first execution of the relevant IaC scripts. We use a Reddit post [8] to explain further. In the post, the user mentions that a new string is appended every time the IaC script is executed. Such execution becomes problematic when the script is run multiple times, because the desired behavior is that the new string will only be added once at the first execution of the script. Idempotency-related defects have not been reported in prior research related to defect categorization for non-IaC software. However, in IaC-related publications, researchers have reported the existence of idempotency defects for Chef scripts [27] and for CFEngine scripts [4].

Example: An idempotency defect caused unwanted changes to artifacts that should not be modified [9]. Idempotency-related defects can cause file backup problems, package update problems, database setup problems, logging problems, and network setup problems [56].

Security: This category represents defects that violate confidentiality, integrity or availability for the provisioned system. Example security-related defects include exposing secrets in logs; specifying SSL certificates that lead to authentication problems; and hard-coding secrets, such as passwords [56]. Prior research on defect categorization has also observed security-related defects to appear in other types of software systems e.g., in blockchain projects [80], video game software [50], cloud management software [84], and OSS Apache, Linux, and Mozilla projects [6] [74].

Example: Setting up user accounts is common in IaC scripts [60], but in the process user confidentiality may be breached. For example, the security defect from Section 1 is listed in a bug report [85]. The defect leaks passwords using variables such as *rabbit_password* in Puppet logs. The defect impacts seven Openstack projects and is fixed by adding the 'secret' attribute [10]. Another example is keeping a token alive indefinitely, which provides access to anyone without authentication and lasts indefinitely if not disabled [11].

Service: This category represents defects related to improper provisioning and inadequate availability of computing services, such as load balancing services and monitoring services. Service-related defects can be caused by improper specification of attributes while using the 'service' resource for Puppet [36], or Chef [8], or the 'service' module for Ansible [3]. Service-related defects have previously reported for cloud management software [84]. Rahman and Williams [60] reported provisioning of web service and database services to appear in defective IaC scripts.

[6] https://bugs.launchpad.net/puppet-neutron/+bug/1288741
[7] https://bugs.launchpad.net/puppet-nova/+bug/1409897
[8] https://www.reddit.com/r/Puppet/comments/679dze/
[9] https://bugs.launchpad.net/fuel/+bug/1572789
[10] https://review.opendev.org/#/c/106529/3/manifests/init.pp
[11] https://bugs.launchpad.net/fuel/+bug/1582893

Example: Service related-defects can cause provisioned services to stop unexpectedly, as it happened for a provisioned MySQL service [12]. The defect occurred as the provisioned service was not working with a title provided as input. Other examples of service-related defects include not being able to start monitoring services, such as Nagios, and load balancing services, such as HAProxy [56].

Syntax: This category represents defects related to syntax in IaC scripts. Syntax-defects have also been reported in prior research studies: Ray et al. [63] and Pascarella et al. [50] in separate studies identified generic programming defects for OSS GitHub projects, which included programming syntax errors. Islam et al. [30] also identified syntax-related defects for deep learning projects.

Example: The Openstack Fuel plugin development workflow [47] uses a continuous integration pipeline, which generates errors if IaC scripts fail style checks. Other examples of syntax-related defects include specifying wrong types for variables [56].

Answer to RQ1: Defect taxonomies can shape tool development, testing and verification efforts, and education about software development. As discussed in Section 2.2, the domain of IaC lacks a defect taxonomy, and thus the ability to prioritize tool development, testing and verification efforts, and disseminate software development knowledge in the classroom could be limited. Answer to RQ1 contributes to the above-mentioned needs by providing (i) a defect taxonomy, and (ii) a discussion related to consequences for each defect category.

3.3 RQ_2: Practitioner Perception

We present the methodology and findings for RQ_2: ***How do practitioners perceive the identified defect categories for infrastructure as code scripts?***

3.3.1 Methodology for RQ_2. Practitioner agreement with the identified defect categories in Section 3.2 can indicate the relevance of the eight categories. We answer RQ_2 by deploying an online survey to practitioners. In the survey, we ask practitioners how many years they worked with IaC scripts. Then, we provide definitions and examples for each category. We wanted to assess if practitioners agree that our categories are in fact IaC-related defect categories. We asked "We have identified eight defect categories for IaC (Puppet) scripts. Each of these categories are listed below. To which extent do you agree with our list?". We construct the survey following Kitchenham and Pfleeger's guidelines [34]: (i) use Likert scale to measure agreement levels: strongly disagree, disagree, neutral, agree, and strongly agree; (ii) add explanations related to the purpose of the study, how to conduct the survey, preservation of confidentiality, and an estimate of completion time; and (iii) conduct a pilot survey to get initial feedback. From the feedback of the pilot survey, we added an open-ended text box so that surveyed practitioners can further respond. The survey questionnaire is available and included in our verifiability package [57].

We offered a drawing of two 50 USD Amazon gift cards as an incentive for participation following Smith et al. [69]'s recommendations on incentivizing surveys. We deploy the survey to 750 practitioners from April 19, 2019 to August 15, 2019. We distributed the survey to practitioners via e-mail following the Internal Review Board (IRB) protocol#16813. We collect practitioner e-mail addresses from the OSS repositories used in our empirical analysis.

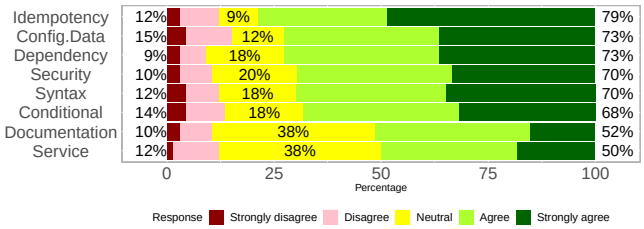

Figure 4: Findings from survey. Practitioners mostly agree with idempotency.

3.3.2 Answer to RQ_2. Of the 750 practitioners, 66 responded. We observe the eight categories to have relevance amongst practitioners: 50%∼79% of the 66 survey respondents agree with one of the eight categories. Respondents agree most with idempotency, with an agreement rate of 79%. The least agreed upon category is service. Details of our survey results is listed in Figure 4, where the defect categories are sorted from top to bottom based on agreement rate. The percentage of respondents who agreed or strongly agreed with each category is listed on the right.

We acknowledge the survey response rate to be low (8.8%). However, low survey response rates are not uncommon in software engineering research: Smith et al. [69] reported software engineering survey response rate to vary from 6% to 36%. From email responses we observe the following reasons for low response: inactive or undeliverable email addresses of practitioners, practitioners asking for full confirmation on receiving monetary incentives, and practitioners not using Puppet in recent years.

Practitioners provide reasoning on why they agree, disagree, or remain neutral. For example, one practitioner considered syntax-related defects to be less important, as syntax-related defects should be found "*during initial testing*". The practitioner identified idempotency and conditional defects as "*two key ones*". Another practitioner highlighted the importance of idempotency, stating idempotency defects "*may not produce functional issues, but can cause product indirect issues*". We notice contrary views related to idempotency as well. One practitioner stated "*I have never run into defects in Puppet's idempotency mechanisms*", instead the practitioner claimed dependency-related defects to be prevalent stating "*Debugging dependency loops is tricky and happens often*". Another practitioner indicated limitations of IaC tools as a possible explanation on why dependency defects could be frequent "*the problem [is] with these [IaC] tools it is not clear how to do it [dependency management]*". For practitioners who provided neutral responses, the presence of defects is more relevant than the category "*I am more concerned about the defect itself rather than its category. If I have that info, it will be good, but it is not mandatory*".

Practitioners also reported defect categories not included in our taxonomy: 'learning curve', 'maintainability', 'parallelism', 'scalability', 'support', and 'usability'. 'Parallelism' and 'scalability' are

[12] https://bugs.launchpad.net/fuel/+bug/1550929

performance-related features of IaC tools. 'Learning curve', 'maintainability', 'support', and 'usability' are related to user experience of IaC tools.

Answer to RQ2: We observe our identified defect categories to have relevance amongst practitioners. Surveyed practitioners have varying opinions on what defect categories are more frequent in IaC development, which are possibly formed by their own experiences. Furthermore, practitioners have reported what additional IaC defect categories may exist that is not included in our taxonomy.

4 AUTOMATED CATEGORIZER FOR INFRASTRUCTURE AS CODE DEFECTS

Catgegorization of IaC defects using raters is resource-consuming. In prior research, Huang et al. [25] emphasized the importance of determining defect categories using an automated technique stating the process of manual software defect categorization as "*at best a mundane, mind numbing activity*". An automated tool to detect IaC defect categories can be useful to (i) analyze repositories with IaC scripts at scale, (ii) mitigate recall bias common in incident reviews [14, 17] e.g., practitioner bias in recalling defects in IaC scripts [13], and (iii) provide groundwork for future defect-related tools for IaC. Researchers can use such automated tool to automatically identify which script includes what defect category.

We construct an automated tool called ***A***utomated ***C***ategorizer for ***I***nfrastructure as Code ***D***efects (***ACID***) to automatically identify IaC defect categories at scale. ACID analyzes each ECM and determines whether or not any of the eight identified defect categories can be identified from the ECM. ACID takes one or multiple repositories as input, and as output reports the defect category for each ECM. If none of the eight categories are identified, ACID reports 'NO DEFECT', indicating no defect is identified. ACID can report an ECM to belong to multiple defect categories. ACID's design is language-independent. ACID uses rules, which uses dependency parsing [32] and pattern matching to detect defect categories with ECMs.

We first describe how we construct the rules used by ACID to detect defect categories. Next, we use a running example to illustrate the components of ACID, and how ACID determines a defect category. Finally, we evaluate the accuracy of ACID by using an oracle dataset.

Rules used by ACID: We construct rules needed for ACID by abstracting patterns that appear in the 1,448 ECMs and their corresponding diffs, obtained from the 61 Openstack OSS repositories. Our hypothesis is that we can abstract patterns from the 1,448 ECMs and their corresponding diffs to automatically detect defect categories for other datasets.

We use Table 1 to demonstrate our approach. The 'ECM' column presents a set of ECMs that include 'fix', a string pattern that represents a defect-related action. The 'Dependent' column shows the string patterns upon which the defect-related action is applied. For example, for 'fix catalog compilation when not configuring endpoint', 'fix' is applied upon 'compilation'. We can abstract patterns using 'fix' and tokens in the 'Dependent' column to construct a rule to identify a defect category. For example, in Table 1, the keywords 'compilation', 'warning', 'lint' and, 'typo' is linked with 'fix'. The first author looked into the set of 1,448 ECMs to determine what string patterns need to be captured as dependents to construct rules.

Table 1: An Example of Identifying Dependents in ECMs for Rule Construction

ECM	Dependent
fix catalog compilation when not configuring endpoint	compilation
fix spurious warning in pipeline check	warning
fix puppet lint 140 characters	lint
typo fix	typo

Some ECMs such as 'Various small fixes', which is downloaded from an OSS repository called 'fuel-plugin-contrail' [14], indicate that a defect-related action occurred but do not clearly express what defect category could be identified. We address this challenge by inspecting diffs of ECMs, in addition to dependency parsing. From the diffs of ECMs we can identify code elements and use them in rules for any of the eight categories. As an example, in Figure 5 we present the changeset for the ECM 'Various small fixes'. We observe that configuration data are changed using three variables '*network_scheme*', '*cidr*', and '*netmask*'.

An ECM can include a sentence that expresses defect-related actions for multiple defect categories. For example, for 'Fix dependencies and various typos', which is downloaded from an OSS repository [48], we observe actions taken to resolve two defect categories: dependency defect and syntax defect. For ECMs with one or multiple sentences, ACID maps an ECM to multiple defect categories if multiple rules are satisfied. The rules ACID uses to detect each of the eight defect categories is listed in Table 2. In Table 2, the 'Category' and 'Rule' column respectively presents the defect categories, and corresponding rules to detect the category. The presented rules use functions. The functions use string pattern matching to check if sentences and/or diffs of an ECM satisfy a certain condition. A mapping of the functions and stemmed string patterns are provided in Table 3. For example, the $hasDefect(x.sen)$ function returns true if any of the provided string patterns listed in Table 3, appear for a sentence in an ECM $x.sen$. In Table 2, functions $changedInclude(x.diff)$, $changedComment(x.diff)$, and $changedService(x.diff)$, identify if the diff of ECM x consists of changes in include statements, comments, and service resources, respectively. $dataChanged(x.diff)$ identifies if the diff includes a change in configuration data. Except for function $hasDefect()$, string patterns for all other functions are derived from our qualitative analysis process. We derive string patterns for $hasDefect()$ from prior work [63] [62], listed in Table 3. For four categories (conditional, idempotency, security, and syntax) ACID does not use diff content because our qualitative analysis did not identify any Puppet-specific code element in the diffs that express the defect category of interest.

Execution of ACID: We construct ECMs using Git, Mercurial, and bug report APIs, and feed the ECMs as input to ACID. ACID

[13] https://community.pagerduty.com/t/incidents-as-we-imagine-them-versus-how-they-actually-are-with-john-allspaw/2708

[14] https://opendev.org/x/fuel-plugin-contrail

```
- $network_scheme = hiera('network_scheme')
- $cidr = $settings['contrail_private_cidr']
- $netmask=cidr_to_netmask($cidr)
- $netmask_short=netmask_to_cidr($netmask)
+ $network_scheme = hiera_hash('network_scheme',{})
+ $cidr = pick(get_network_role_property('neutron/mesh','cidr'),
    ↪ get_network_role_property('contrail/vhost0','cidr'))
+ $netmask = pick(get_network_role_property('neutron/mesh','
    ↪ netmask'),get_network_role_property('contrail/vhost0','
    ↪ netmask'))
+ $netmask_short = netmask_to_cidr($netmask)
```

Figure 5: Diff of ECM 'Various small fixes'. Configuration data are changed in the diff indicating the ECM to be related with category 'configuration data'.

Table 2: Rules to Detect Defect Categories

Category	Rule
Conditional	$hasDefect(x.sen) \wedge hasCond(x.sen.dep)$
Configuration Data	$hasDefect(x.sen) \wedge ((hasStorConf(x.sen.dep) \vee hasFileConf(x.sen.dep) \vee hasNetConf(x.sen.dep) \vee hasUserConf(x.sen.dep) \vee hasCachConf(x.sen.dep)) \vee dataChanged(x.diff))$
Dependency	$hasDefect(x.sen) \wedge (hasDepe(x.sen.dep) \vee changedInclude(x.diff))$
Documentation	$hasDefect(x.sen) \wedge (hasDoc(x.sen.dep) \vee changedComment(x.diff))$
Idempotency	$hasDefect(x.sen) \wedge hasIdem(x.sen.dep)$
Security	$hasDefect(x.sen) \wedge hasSecu(x.sen.dep)$
Service	$hasDefect(x.sen) \wedge (hasServ(x.sen.dep) \vee changedService(x.diff))$
Syntax	$hasDefect(x.sen) \wedge hasSynt(x.sen.dep)$

Table 3: String Patterns Used for Functions in Rules

Function	String Pattern
$hasDefect()$	'error', 'bug', 'fix', 'issu', 'mistake', 'incorrect', 'fault', 'defect', 'flaw'
$hasCond()$	'logic', 'condit', 'boolean'
$hasStorConf()$	'sql', 'db', 'databas'
$hasFileConf()$	'file', 'permiss'
$hasNetConf()$	'network', 'ip', 'address', 'port', 'tcp', 'dhcp'
$hasUserConf()$	'user', 'usernam', 'password'
$hasCachConf()$	'cach'
$hasDepe()$	'requir', 'depend', 'relat', 'order', 'sync', 'compat', 'ensur', 'inherit'
$hasDoc()$	'doc', 'comment', 'spec', 'licens', 'copyright', 'notic', 'header', 'readm'
$hasIdem()$	'idempot'
$hasSecu()$	'vulner', 'ssl', 'secr', 'authent', 'password', 'secur', 'cve'
$hasServ()$	'servic', 'server'
$hasSynt()$	'compil', 'lint', 'warn', 'typo', 'spell', 'indent', 'regex', 'variabl', 'whitespac'

uses four steps to identify a defect category: sentence tokenization, text pre-processing, dependency parsing, and rule matching. We use Table 4 to describe how ACID processes an example ECM "*Update incorrect comment about nova-network status. The network manifest contained a comment that said that nova-network was no longer receiving any patches. That's not actually the case; so replace the comment with a new description that describes the current state*", which is downloaded from the 'puppet-nova' repository [15]. The ECM also includes a change in comment, as shown in Figure 6.

[15]https://opendev.org/openstack/puppet-nova

```
 # [*enabled*]
 #   (optional) Whether the network service should be enabled.
-#   Defaults to false
+#   Defaults to true
```

Figure 6: Diff of ECM used as running example where source code comments are changed.

Table 4: Example to Demonstrate ACID's Execution

Step #1	Step #2	Step #3	Step #4
Update incorrect comment about nova-network status	updat incorrect comment about nova network statu	HEAD:[updat], DEPE:[comment]	Rule matched for documentation defect as per Table 2
The network manifest contained a comment that said that nova-network was no longer receiving any patches.	network manifest contain comment that said that nova network was no longer receiv ani patch	HEAD:[contain], DEPE:[comment]	No match
That's not actually the case; so replace the comment with a new description that describes the current state.	that not actual case so replac comment with new descript that describ current state	HEAD:[replac], DEPE:[comment]	No match

Step #1-Sentence tokenization: We apply sentence tokenization [33] on each ECM to split an ECM into multiple sentences. In the case of Table 4, the ECM includes three sentences, and upon application of Step #1, we obtain three individual sentences: (i) 'Update incorrect comment about nova-network status', (ii) 'The network manifest contained a comment that said that nova-network was no longer receiving any patches', and (iii) 'That's not actually the case; so replace the comment with a new description that describes the current state'.

Step#2-Text Pre-processing: From each ECM, we remove English stop words, such as 'a', 'the' and 'by'. Next, we remove special characters and numeric literals. Then, we apply porter stemming [53] on each word for each ECM. ACID will apply text pre-processing for each of the individual sentences for the ECM provided in Table 4. The output of text pre-processing is provided in the 'Step #2' column.

Step#3-Dependency parsing: Using dependency parsing [32] we identify heads and dependents for output obtained from Step#2. Dependency parsing identifies a 'head', an action item, and 'dependents' i.e. the words in the sentence which are dependent upon the action item. For example, in Table 4 we observe the keyword 'updat' to be a head. The dependent for 'updat' is 'comment'. To identify dependents and heads, we use the Spacy API [70], which leverages datasets, where heads and dependents are annotated by human raters for the English language [32]. Using this annotation, dependency parsing identifies what tokens in a sentence are heads and dependents. We use the identified heads and dependents in the next step. Output after Step #3 for our example ECM is listed in 'Step #3' of Table 4.

Step#4-Rule Matching: From dependency parsing output of Step#3, we apply rule matching to determine a defect category. The rules are listed in Table 2. If a sentence for an ECM satisfies any of the rules, the corresponding defect category is assigned to that ECM. For our running example, we observe the rule '$hasDefect(x.sen)$

Table 5: ACID's Accuracy for Oracle Dataset

Category	Occurr.	Precision	Recall
Conditional	6	0.75	1.00
Configuration Data	29	0.91	1.00
Dependency	3	0.75	1.00
Documentation	6	0.75	1.00
Idempotency	3	0.75	1.00
Security	1	1.00	1.00
Service	13	0.85	0.85
Syntax	6	0.86	1.00
No defect	65	0.96	0.82
Average		0.84	0.96

$\wedge$ ($hasDoc(x.sen.dep) \vee changedComment(x.diff)$)' is satisfied as (i) the ECM satisfies '$hasDefect(x.sen)$', as the keyword 'incorrect' appears in 'updat incorrect comment about nova network statu', (ii) the sentence 'updat incorrect comment about nova network statu' includes a dependent 'comment', which satisfies '$hasDoc(x.sen.dep)$', and (iii) as shown in Figure 6, comments are changed in the diff for the ECM, so '$changedComments(x.diff)$' is satisfied. For the other sentences no rules are matched ('No match' in column 'Step#4'). Therefore, the ECM belongs to 'documentation'.

Evaluation of ACID: We use raters who are not authors of the paper to construct an oracle dataset to evaluate ACID. To construct the oracle dataset, raters perform closed coding [65], where they map each assigned ECM to any of the categories identified in Section 3.2. The mapping task was made available using a website [16]. Each rater was provided the IEEE Standard Classification for Software Anomalies [28] and a handbook on Puppet [36] for reference.

We recruit raters from a graduate-level class of 60 students, where 22 students volunteered to participate in constructing the oracle dataset. The graduate class is focused on DevOps principles. We used balanced incomplete block design [18] to select and distribute the 132 ECMs amongst the 22 raters, so that each ECM is reviewed by at least two raters. Upon completion of the mapping task, we observe the agreement rate to be 71.9%, and Cohen's Kappa to be 0.61, which is 'substantial agreement' according to Landis and Koch [37]. The first author resolved disagreements.

Upon completion of constructing the oracle dataset, we evaluate ACID by computing the precision and recall of ACID for the oracle dataset. Precision refers to the fraction of correctly-identified categories among the total identified defect categories, as determined by ACID. Recall refers to the fraction of correctly-identified defect categories that have been retrieved by ACID over the total amount of defect categories. We report the precision and recall values for each defect category in Table 5. We observe the average precision and recall to be respectively 0.84 and 0.96 across all categories.

We notice that false negatives occur when the dependency parsing technique incorrectly identify tokens in the ECM that have no relationship to a defect category or identifies tokens that are related to an incorrect category. For example, for the ECM 'fix following warnings', instead of 'warnings' dependency parsing incorrectly identifies 'following' as the dependent. We observe keyword matching to contribute to false positives, for example ACID incorrectly identifies the ECM 'Bug 1085520 - Support instance-store backed AMIs for builders. Add new secrets.' as a security-related defect, even though the ECM corresponds to adding a new feature.

Verifiability: All constructed datasets and ACID's source code are available online [57]. ACID is also available as a Docker image for use [17].

[16] http://13.59.115.46/joybangla/login.php

Table 6: OSS Repositories Satisfying Curation Criteria

	GHB	MOZ	OST	WIK
Initial Repo. Count	14,856,957	1,858	2,120	2,031
Criteria-1 (11% IaC Scripts)	6,088	2	67	11
Criteria-2 (Not a Clone)	4,040	2	61	11
Criteria-3 (Commits/Month $\geq$ 2)	2,710	2	61	10
Criteria-4 (Contributors $\geq$ 10)	218	2	61	10
Final Repo. Count	218	2	61	10

5 EMPIRICAL ANALYSIS

In this section, we provide the methodology and findings for the research question: **RQ$_3$**: *How frequently do the identified defect categories appear for infrastructure as code scripts?*

Table 7: Attributes of the Four Datasets

Attribute	GHB	MOZ	OST	WIK
Repo. Type	Git	Mercurial	Git	Git
Tot. Repos	218	2	61	10
Tot. Commits	434,234	14,449	44,469	71,795
Tot. Puppet Scripts	10,025	1,596	2,845	3,143
Tot. Puppet-related Commits	40,286	6,836	12,227	21,066
Time Period	01/2005-04/2019	05/2011-04/2019	09/2011-04/2019	01/2006-04/2019

5.1 Datasets

We conduct our empirical analysis with four datasets of Puppet scripts. We construct three datasets from the OSS repositories maintained by three organizations: Mozilla [42], Openstack [45], and Wikimedia Commons [13]. We select repositories from these three organizations because these organizations create or use cloud-based services. We construct the other dataset from OSS repositories hosted on GitHub, as companies tend to host their popular OSS projects on GitHub [35] [1]. In our collected repositories IaC-related ECMs correspond to defects that have been previously reported by respective practitioners.

Munaiah et al. [43] advocated for curation of OSS repositories before conducting empirical analysis. We apply the following criteria to curate the collected repositories: *Criteria-1*: At least 11% of the files belonging to the repository must be IaC scripts. Jiang and Adams [31] reported that in OSS repositories IaC scripts co-exist with other types of files, such as 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 repositories that contain sufficient amount of IaC scripts. *Criteria-2*: The repository is not a copy of another repository. *Criteria-3*: The repository must have at least two commits per month. Munaiah et al. [43] used the threshold of at least two commits per month to determine which repositories have enough software development activity. We use this threshold to filter repositories with limited activity. *Criteria-4*: The repository has at least 10 contributors. Our assumption is that the criteria of

[17] https://hub.docker.com/r/akondrahman/acid-puppet

Table 8: Sanity Checking ACID's Accuracy

Category	Occurr.	Precision	Recall
Conditional	15	0.88	0.93
Configuration Data	99	0.90	0.96
Dependency	59	0.86	0.93
Documentation	32	0.91	0.94
Security	23	0.88	0.91
Service	54	0.85	0.93
Syntax	74	0.93	0.93
No defect	1644	0.99	0.98
Average		0.90	0.94

at least 10 contributors may help us to filter out irrelevant repositories. Previously, researchers have used the cutoff of at least nine contributors [55] [1] as a curation criterion. As shown in Table 6, 291 repositories met our filtering criteria. Attributes of the collected repositories are available in Table 7.

5.2 Sanity Check for ACID

Before applying ACID for all datasets, we apply a sanity check to assess ACID's detection performance for ECMs not included in the oracle dataset. The first author randomly selected 500 ECMs from each of the four datasets, and applied ACID on 2,000 ECMs. We report the precision and recall in Table 8. We observe the recall for all defect categories is >= 0.91, which indicates ACID's ability to detect most existing defect categories, but generate false positives.

5.3 RQ_3: Frequency of Defect Categories

In this section, we answer ***How frequently do the identified defect categories appear in infrastructure as code scripts?***.

5.3.1 Methodology for Defect Category Frequency. We answer RQ_3 by reporting 'defect proportion' and 'script proportion' values, as well as, temporal frequency for each defect category. 'Defect Proportion' refers to the percentage of commits in the dataset that belong to a defect category, similar to Hindle et al. [24]. The defect proportion metric provides a summarized view of defect category frequency across the whole dataset, whereas, 'script proportion' refers to the proportion of scripts that are modified in commits that include at least one category. Practitioners can find 'script proportion' values useful, as the metric quantifies how many scripts include at least one defect category. 'Defect/Year' shows how each category of defects evolve with time, and summarizes temporal trends. We report the temporal frequency for each year using Equation 1, where we calculate the metric 'Defect/Year' for a certain category x that occurs in year y.

$$\text{Defect/Year}(x, y)\% = \frac{\text{\# of ECMs in year } y \text{ and marked as defect category } x}{\text{total ECMs in year } y} * 100\% \quad (1)$$

5.3.2 Answer to RQ_3: How frequently do the identified defect categories appear in infrastructure as code scripts? We report defect proportion and script proportion values for the eight categories in Table 9. Configuration data defects is the most frequently occurring category based on the defect proportion metric. Network is the most frequent subcategory: 75.3%~88.2% of the identified configuration data defects are network defects. Frequency of subcategories for configuration data defects is available online [57]. We observe 23.5%~33.9% scripts to be modified in commits related to configuration data, which is the highest across all defect categories. As reported in the 'Total' row in Table 9, we observe 16.6%, 10.9%, 18.7%, and 15.2% of all commits to include at least one of the eight defect categories, respectively, for GitHub, Mozilla, Openstack, and Wikimedia. Also, we observe 43.6%, 47.4%, 48.6%, and 49.2% of all scripts to include at least one defect category respectively, for GitHub, Mozilla, Openstack, and Wikimedia. Similar to defect proportion, based on script proportion values, we observe configuration data-related defects to be the dominant category across all four datasets.

Table 9: Defect and Script Proportion for Defect Categories

	Defect Prop. (%)				Script Prop. (%)			
Categ.	**GHB**	**MOZ**	**OST**	**WIK**	**GHB**	**MOZ**	**OST**	**WIK**
Cond.	0.3	0.04	0.3	0.1	1.9	0.3	1.8	1.4
Conf.Data	9.5	3.8	11.5	7.2	29.2	23.5	33.9	29.6
Depe.	1.8	1.3	2.4	1.7	10.6	12.4	16.3	17.1
Docu.	1.7	0.8	1.6	1.5	13.7	7.2	14.0	13.6
Idem.	0.1	0.01	0.3	0.01	1.0	1.6	3.5	0.1
Secu.	0.1	0.5	0.5	0.1	0.9	5.5	2.8	1.1
Serv.	1.4	2.6	1.8	3.0	4.9	23.1	9.3	12.4
Synt.	2.8	1.7	2.3	2.4	16.3	9.2	19.2	17.4
Total	16.6	10.9	18.7	15.2	43.6	47.4	48.6	49.2

In Figure 7, the x-axis presents years, and the y-axis presents defect/year values for each year. From Figure 7, over time we observe defects to be reported across all datasets as indicated by 'TOTAL'. For configuration data-related defects, the defect/year value does not reduce to zero, indicating configuration data defects to be reported throughout the entire lifetime period for all four datasets.

Our defect categories can correlate. Correlating categories are detectable: if rules for multiple categories are satisfied then ACID will report an ECM belonging to multiple categories. ECMs that tested positive for two categories were 1.01%, 0.05%, 1.72%, and 0.82%, respectively, for GitHub, Mozilla, Openstack, and Wikimedia. ECMs that tested positive for three categories were 0.09%, 0.00%, 0.12%, and 0.04%, respectively, for GitHub, Mozilla, Openstack, and Wikimedia. For any of the datasets we do not observe ECMs that tested positive for four or more defect categories.

Answer to RQ3: Configuration data is the most dominant defect category. Our identified defect categories can correlate, for example, ECMs that tested positive for two categories were 0.05%~8.01% across four datasets.

6 DISCUSSION

We discuss our findings in this section.

Practitioner Perception and Observed Evidence: According to Srikanth and Menzies [5], "*documenting developer beliefs should be the start, not the end, of software engineering research. Once prevalent beliefs are found, they should be checked against real-world data*", suggesting researchers to complement survey data with software repository data. We too have complemented survey data with analysis of OSS data. We have reported varying practitioner perceptions for the identified categories in Section 3.3.2. We notice congruence

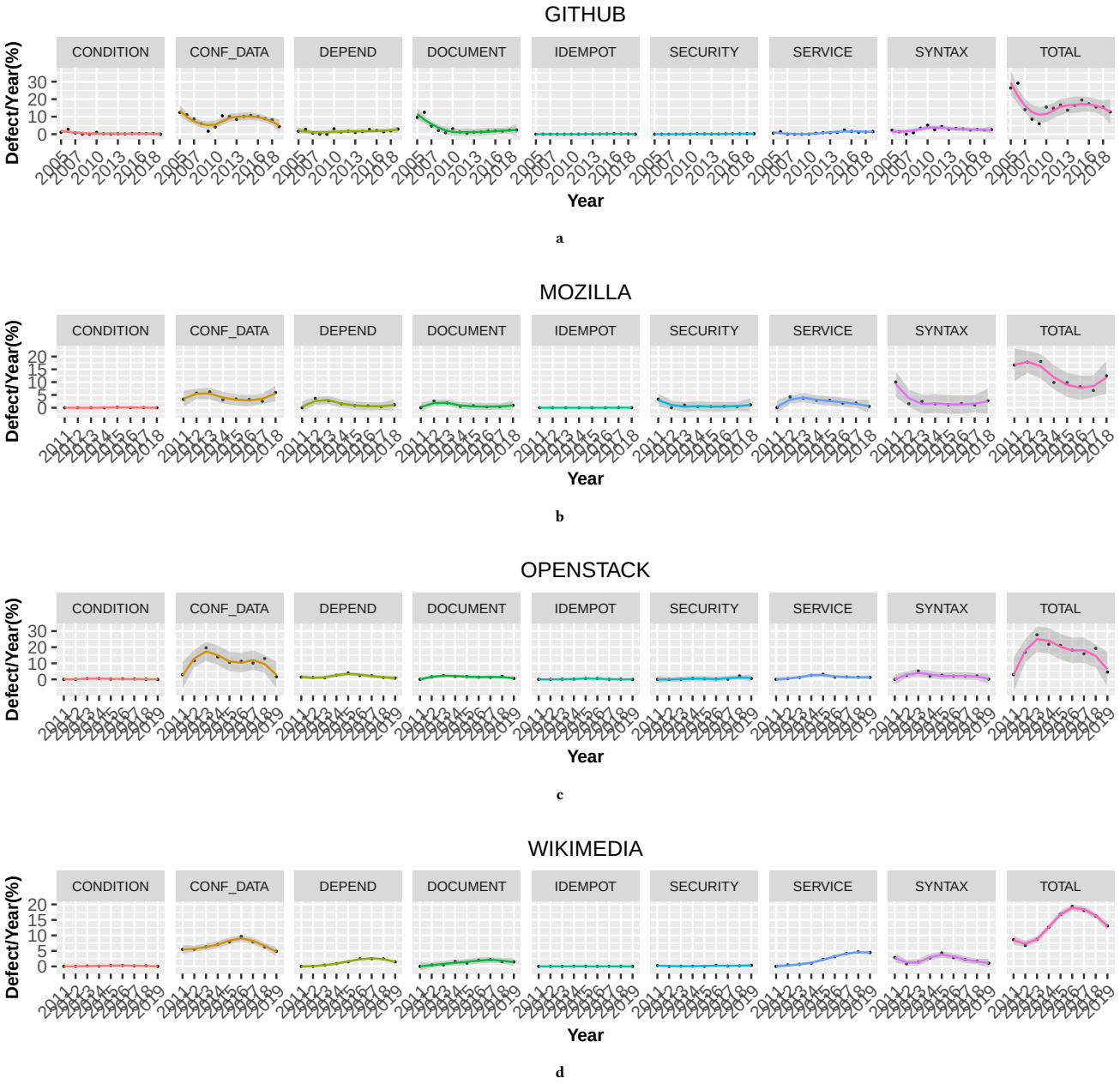

Figure 7: Evolution of defect proportion for eight categories; GitHub: 7a, Mozilla: 7b, Openstack: 7c, and Wikimedia: 7d. For each dataset, 'Total' presents the proportion of commits, which includes at least one category of defect.

for two categories: configuration data and dependency are the second most agreed upon category and also frequently occurs in OSS datasets. On the other hand, service defects are least agreed upon, but they are more frequent than idempotency, the defect category surveyed practitioners most agreed upon.

In the Reddit post [16] mentioned in Section 1, practitioners reported only one defect category–syntax. Along with syntax, our taxonomy includes seven other defect categories. Suggestions from online forums could be inconclusive, and practitioners can find our taxonomy helpful.

Mitigation: Companies can mitigate the occurrence of defects by incorporating tools that target one or more of the identified categories during IaC development. For example, companies can adopt 'rspec-puppet' [64] to reduce conditional and service defects.

Tools such as Tortoise [81], 'librarian-puppet' [38], and 'puppet-strings' [29] might be helpful respectively, in mitigating configuration data, dependency, and documentation defects. Static analysis tools such as SLIC [58] and 'puppet-lint' [54] might respectively, be helpful in mitigating security and syntax defects. Mitigation of idempotency defects might be possible through early detection of idempotency with Hummer et al. [27]'s approach that uses model-based testing [77] to detect idempotency.

Implications: Our paper can be helpful in the following manner:

- *better understanding of defects*: use of definitions and examples in Section 3.2 to understand the consequences of IaC defect categories, and activities needed to mitigate each defect category;
- *triaging defects*: using our taxonomy, practitioners can find what IaC-related defect categories are being reported for a long period, helping them make informed decisions on defect triaging;
- *measuring IaC script quality*: use our reported frequency in Table 9 as a reference;
- *saving manual effort*: Zou et al. [86] identified three reasons why practitioners find automated defect categorization important: better resource allocation, saving manual work in classifying defects, and facilitating postmortem analysis. For automated categorization of IaC defects ACID can be helpful for practitioners. Manual analysis of one ECM on average has taken 75 seconds per rater, whereas, ACID takes 0.09 seconds on a 'macOS Mojave' laptop with 1.4 GHz Intel Core i5 processor and 16 GB memory. ACID could be useful for teams that don't have raters to perform postmortem analysis using qualitative coding; and
- *constructing IaC-related education materials*: educators who conduct IaC-related courses at the undergraduate and graduate level, can use Section 3.2 to showcase what types of quality issues can arise while developing IaC scripts.

Future Work: Researchers can investigate if above-mentioned recommendations can actually reduce defects in IaC scripts. The coding patterns that ACID use, could be further leveraged in investigating if defect categories for IaC, such as configuration data, can be detected at compile time.

7 THREATS TO VALIDITY

We briefly describe the limitations of our paper in this section.

Conclusion Validity: The derived defect categories and the oracle dataset are susceptible to rater bias. In both cases, we mitigate rater bias by allocating multiple raters. Also, we use the content of commit messages to determine the defect categories for IaC scripts, which is limiting as commit messages do not always express a defect-related action. Practitioners may use other keywords that we did not include. We mitigate this limitation by using a set of defect-related keywords, derived from prior research, and shown to be adequate in detecting defect-related commits [63] [62] [41]. Also, ACID uses dependency parsing that relies on annotated datasets mined from news articles [32], which can be limiting to capture dependencies.

External Validity: We have not analyzed proprietary repositories, and our findings are limited to OSS datasets with Puppet scripts. We mitigate this limitation by mining OSS repositories from GitHub and three organizations. We conduct our empirical study with one IaC tool called Puppet. We acknowledge that our findings may be limited to Puppet. However, evidence demonstrates our categories to exist across languages: e.g., idempotency appears for Chef [27] and CFEngine [4]. Considering frequency and category, Schwarz et al. [66] found configuration-related code smells to generalize across multiple languages, which suggests that our findings related to configuration data defects may generalize too. Furthermore, ACID's design is language-independent. ACID uses dependency parsing and pattern matching to detect defect categories.

Internal Validity: The defect category list is not comprehensive, as the derivation process is dependent on the collected commits and rater judgment. We mitigate this limitation with 1,448 ECMs and two raters to derive defect categories. We acknowledge the limitations of the rules presented in Table 2, as the construction is dependent upon the ECMs and diffs of the collected commits, along with the first author's judgment. Practitioners can use certain string patterns to describe a category that we did not list. We mitigate this limitation by inspecting 1,448 ECMs and diffs to derive the used string patterns.

8 CONCLUSION

Defects in IaC scripts can have serious consequences. Defect categorization can help practitioners to make informed decisions on how to mitigate IaC defects. We applied qualitative analysis on 1,448 defect-related commits to identify defect categories for IaC scripts. We surveyed 66 practitioners to assess if practitioners agree with the identified defect categories. Next, we constructed a tool called ACID and apply ACID on 80,415 commits from 291 repositories to automatically identify defect categories in IaC scripts.

Our derived taxonomy consists of eight defect categories that included idempotency, a category specific to IaC. Amongst 66 survey respondents, 79% of the practitioners agreed with idempotency-related defects, and 49%∼79% of the practitioners agreed with one of the identified defect categories. We observe configuration data-related defects to be the most frequent defect category, whereas idempotency is the least frequently occurring defect category. Using our reported defect category frequency results, practitioners can prioritize V&V efforts by fixing configuration data defects that occur in 23.5%∼33.9% of IaC scripts.

Our research can be helpful for practitioners to improve IaC script quality, as they can use ACID to identify defect categories automatically, and use our definitions and examples of each identified defect category to assess the importance of the identified defect categories. Our taxonomy of IaC defects can help practitioners in understanding the nature of defects and guide them in triaging defects, prioritizing V&V efforts, and measuring IaC script quality. We hope our findings will facilitate further research in the domain IaC script quality.

ACKNOWLEDGMENTS

We thank the anonymous practitioners for participating in our survey. We thank the RealSearch group at NC State University and the anonymous reviewers for their valuable feedback. Our research was partially funded by the NSA's Science of Security Lablet at NC State University.

REFERENCES

[1] Amritanshu Agrawal, Akond Rahman, Rahul Krishna, Alexander Sobran, and Tim Menzies. 2018. We Don't Need Another Hero?: The Impact of "Heroes" on Software Development. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18)*. ACM, New York, NY, USA, 245–253. https://doi.org/10.1145/3183519.3183549

[2] Marc Ambasna-Jones. 2018. Automation key to unravelling mysteries of the universe at CERN. https://www.computerweekly.com/feature/Automation-key-to-unravelling-mysteries-of-the-universe-at-CERN. [Online; accessed 17-August-2019].

[3] Ansible. 2019. Ansible Documentation. https://docs.ansible.com/. [Online; accessed 19-August-2019].

[4] Mark Burgess. 2011. Testable System Administration. *Commun. ACM* 54, 3 (March 2011), 44–49. https://doi.org/10.1145/1897852.1897868

[5] Shrikanth N. C. and Tim Menzies. 2019. Assessing Developer Beliefs: A Reply to "Perceptions, Expectations, and Challenges in Defect Prediction". *CoRR* abs/1904.05794 (2019). arXiv:1904.05794 http://arxiv.org/abs/1904.05794

[6] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152 (2019), 165 – 181. https://doi.org/10.1016/j.jss.2019.03.002

[7] CERN. 2018. Key Facts and Figures CERN Data Centre. http://information-technology.web.cern.ch/sites/information-technology.web.cern.ch/files/CERNDataCentre_KeyInformation_01June2018V1.pdf. [Online; accessed 17-August-2019].

[8] Chef. 2019. Chef Docs. https://docs.chef.io/. [Online; accessed 19-August-2019].

[9] Ram Chillarege, Inderpal Bhandari, Jarir Chaar, Michael Halliday, Diane Moebus, Bonnie Ray, and Man-Yuen Wong. 1992. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering* 18, 11 (Nov 1992), 943–956. https://doi.org/10.1109/32.177364

[10] Marcelo Cinque, Dominico Cotroneo, Raffaele D. Corte, and Antonio Pecchia. 2014. Assessing Direct Monitoring Techniques to Analyze Failures of Critical Industrial Systems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 212–222. https://doi.org/10.1109/ISSRE.2014.30

[11] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. https://doi.org/10.1177/001316446002000104 arXiv:http://dx.doi.org/10.1177/001316446002000104

[12] Wikimedia Commons. 2017. Incident documentation/20170118-Labs. https://wikitech.wikimedia.org/wiki/Incident_documentation/20170118-Labs. [Online; accessed 27-Jan-2019].

[13] Wikimedia Commons. 2019. Project: Gerrit Wikimedia. https://gerrit.wikimedia.org/r/#/admin/projects/

[14] Richard I Cook. 1998. How complex systems fail. *Cognitive Technologies Laboratory, University of Chicago. Chicago IL* (1998).

[15] Domenico Cotroneo, Roberto Pietrantuono, and Stefano Russo. 2013. Testing Techniques Selection Based on ODC Fault Types and Software Metrics. *J. Syst. Softw.* 86, 6 (June 2013), 1613–1637. https://doi.org/10.1016/j.jss.2013.02.020

[16] Csebuetian. 2019. Defect Categories for Puppet Scripts. https://www.reddit.com/r/Puppet/comments/becwq3/defect_categories_for_puppet_scripts/. [Online; accessed 20-August-2019].

[17] Sidney W. A. Dekker. 2002. Reconstructing human contributions to accidents: the new view on error and performance. *Journal of safety research* 33.3 (2002), 371–85.

[18] Joseph L. Fleiss. 1981. Balanced Incomplete Block Designs for Inter-Rater Reliability Studies. *Applied Psychological Measurement* 5, 1 (1981), 105–112. https://doi.org/10.1177/014662168100500115 arXiv:https://doi.org/10.1177/014662168100500115

[19] James Fryman. 2014. DNS Outage Post Mortem. https://github.blog/2014-01-18-dns-outage-post-mortem/. [Online; accessed 20-August-2019].

[20] Jeanne Gu. 2019. GM Extract, Transform and Load Platform as a Service. https://pages.chef.io/rs/255-VFB-268/images/Chef_GM_ETL_Platform_as_a_Service.pdf. [Online; accessed 18-August-2019].

[21] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. 2016. Asserting Reliable Convergence for Configuration Management Scripts. *SIGPLAN Not.* 51, 10 (Oct. 2016), 328–343. https://doi.org/10.1145/3022671.2984000

[22] Russel Harrison. 2013. How to Avoid Puppet Dependency Nightmares With Defines. https://blog.openshift.com/how-to-avoid-puppet-dependency-nightmares-with-defines/. [Online; accessed 22-Jul-2019].

[23] Rebecca Hersher. 2017. Incident documentation/20170118-Labs. https://www.npr.org/sections/thetwo-way/2017/03/03/518322734/amazon-and-the-150-million-typo. [Online; accessed 27-Jan-2019].

[24] Abram Hindle, Daniel M. German, and Ric Holt. 2008. What Do Large Commits Tell Us?: A Taxonomical Study of Large Commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR '08)*. ACM, New York, NY, USA, 99–108. https://doi.org/10.1145/1370750.1370773

[25] Liguo Huang, Vincent Ng, Isaac Persing, Mingrui Chen, Zeheng Li, Ruili Geng, and Jeff Tian. 2015. AutoODC: Automated Generation of Orthogonal Defect Classifications. *Automated Software Engg.* 22, 1 (March 2015), 3–46. https://doi.org/10.1007/s10515-014-0155-1

[26] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* (1st ed.). Addison-Wesley Professional.

[27] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Testing Idempotence for Infrastructure as Code. In *Middleware 2013*, David Eyers and Karsten Schwan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 368–388.

[28] IEEE. 2010. IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* (Jan 2010), 1–23. https://doi.org/10.1109/IEEESTD.2010.5399061

[29] Puppet Inc. 2019. puppet-strings. https://rubygems.org/gems/puppet-strings. [Online; accessed 22-Aug-2019].

[30] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510âĂŞ520. https://doi.org/10.1145/3338906.3338955

[31] Yujuan Jiang and Bram Adams. 2015. Co-evolution of Infrastructure and Source Code: An Empirical Study. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 45–55. http://dl.acm.org/citation.cfm?id=2820518.2820527

[32] Daniel Jurafsky and James H. Martin. 2009. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[33] Bryan Jurish and Kay-Michael Wurzner. 2013. Word and Sentence Tokenization with Hidden Markov Models. *JLCL* 28, 2 (2013), 61–83.

[34] Barbara A. Kitchenham and Shari L. Pfleeger. 2008. *Personal Opinion Surveys*. Springer London, London, 63–92. https://doi.org/10.1007/978-1-84800-044-5_3

[35] Rahul Krishna, Amritanshu Agrawal, Akond Rahman, Alexander Sobran, and Tim Menzies. 2018. What is the Connection Between Issues, Bugs, and Enhancements?: Lessons Learned from 800+ Software Projects. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18)*. ACM, New York, NY, USA, 306–315. https://doi.org/10.1145/3183519.3183548

[36] Puppet Labs. 2017. Puppet Documentation. https://docs.puppet.com/. [Online; accessed 19-August-2019].

[37] Richard Landis and Gary Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174. http://www.jstor.org/stable/2529310

[38] Librarian puppet. 2019. librarian-puppet. http://librarian-puppet.com/. [Online; accessed 22-Aug-2019].

[39] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velasquez. 2017. An Empirical Study on Android-related Vulnerabilities. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, Piscataway, NJ, USA, 2–13. https://doi.org/10.1109/MSR.2017.60

[40] James Turnbull McCune and Jeffrey. 2011. *Pro Puppet* (1 ed.). Apress. 336 pages. https://doi.org/10.1007/978-1-4302-3058-8

[41] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes Using Historic Databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00) (ICSM '00)*. IEEE Computer Society, Washington, DC, USA, 120–. http://dl.acm.org/citation.cfm?id=850948.853410

[42] Mozilla. 2019. Mercurial Repositories Index. https://hg.mozilla.org/

[43] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* (2017), 1–35. https://doi.org/10.1007/s10664-017-9512-6

[44] Openstack. 2018. OpenStack git repository browser. http://git.openstack.org/cgit/. [Online; accessed 12-September-2018].

[45] Openstack. 2019. Explore-OpenDev. https://opendev.org/explore/repos

[46] Openstack. 2019. fuel-plugin-lma-collector. https://opendev.org/x/fuel-plugin-lma-collector. [Online; accessed 22-August-2019].

[47] Openstack. 2019. Openstack Fuel Plugin. https://wiki.openstack.org/wiki/Fuel/Plugins#Example_jobs. [Online; accessed 19-August-2019].

[48] Openstack. 2019. puppet-ceilometer. https://opendev.org/openstack/puppet-ceilometer. [Online; accessed 23-August-2019].

[49] Openstack. 2019. Welcome to Neutron's documentation! https://docs.openstack.org/neutron/latest/. [Online; accessed 22-August-2019].

[50] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. 2018. How is Video Game Development Different from Software Development in Open Source?. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, New York, NY, USA, 392–402. https://doi.org/10.1145/3196398.3196418

[51] Antonio Pecchia and Stefano Russo. 2012. Detection of Software Failures through Event Logs: An Experimental Study. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. 31–40. https://doi.org/10.1109/ISSRE.2012.24

[52] Google Cloud Platform. 2018. Google Compute Engine Incident #17007. https://status.cloud.google.com/incident/compute/17007#5659118702428160. [Online;

accessed 20-August-2019].

[53] MF Porter. 1997. Readings in Information Retrieval. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter An Algorithm for Suffix Stripping, 313–316. http://dl.acm.org/citation.cfm?id=275537.275705

[54] Puppet lint. 2019. puppet-lint. http://puppet-lint.com/. [Online; accessed 22-Aug-2019].

[55] Akond Rahman, Amritanshu Agrawal, Rahul Krishna, and Alexander Sobran. 2018. Characterizing the Influence of Continuous Integration: Empirical Results from 250+ Open Source and Proprietary Projects. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics (SWAN 2018)*. ACM, New York, NY, USA, 8–14. https://doi.org/10.1145/3278142.3278149

[56] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. 2019. References to Defect-related Consequences. http://tiny.cc/defect-result-source. [Online; accessed 22-Aug-2019].

[57] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. 2019. Verifiability Package for Paper. https://figshare.com/s/b2633bd4b1929267a451

[58] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure As Code Scripts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 164–175. https://doi.org/10.1109/ICSE.2019.00033

[59] Akond Rahman, Asif Partho, Patrick Morrison, and Laurie Williams. 2018. What Questions Do Programmers Ask About Configuration As Code?. In *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering (RCoSE '18)*. ACM, New York, NY, USA, 16–22. https://doi.org/10.1145/3194760.3194769

[60] Akond Rahman and Laurie Williams. 2018. Characterizing Defective Configuration Scripts Used for Continuous Deployment. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 34–45. https://doi.org/10.1109/ICST.2018.00014

[61] Akond Rahman and Laurie Williams. 2019. Source Code Properties of Defective Infrastructure as Code Scripts. *Information and Software Technology* (2019). https://doi.org/10.1016/j.infsof.2019.04.013

[62] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 428–439. https://doi.org/10.1145/2884781.2884848

[63] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 155–165. https://doi.org/10.1145/2635868.2635922

[64] Rspec puppet. 2019. rspec-puppet. https://rspec-puppet.com/. [Online; accessed 22-Aug-2019].

[65] Johnny Saldana. 2015. *The coding manual for qualitative researchers*. Sage.

[66] J. Schwarz, A. Steffens, and H. Lichter. 2018. Code Smells in Infrastructure as Code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. 220–228. https://doi.org/10.1109/QUATIC.2018.00040

[67] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A Configuration Verification Tool for Puppet. *SIGPLAN Not.* 51, 6 (June 2016), 416–430. https://doi.org/10.1145/2980983.2908083

[68] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does Your Configuration Code Smell?. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 189–200. https://doi.org/10.1145/2901739.2901761

[69] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. 2013. Improving developer participation rates in surveys. In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 89–92. https://doi.org/10.1109/CHASE.2013.6614738

[70] SpaCy. 2019. spaCy: Industrial-Strength Natural Language Processing. https://spacy.io/. [Online; accessed 20-August-2019].

[71] StackExchange. 2019. Tour-Stack Exchange. https://stackexchange.com/tour. [Online; accessed 22-August-2019].

[72] StackExchange Status. 2014. Outage Post-Mortem: August 25th, 2014. https://stackstatus.net/post/96025967369/outage-post-mortem-august-25th-2014. [Online; accessed 20-August-2019].

[73] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. 2008. TODO or to Bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 251–260. https://doi.org/10.1145/1368088.1368123

[74] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug Characteristics in Open Source Software. *Empirical Softw. Engg.* 19, 6 (Dec. 2014), 1665–1705. https://doi.org/10.1007/s10664-013-9258-8

[75] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*Icomment: Bugs or Bad Comments?*/. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 145–158. https://doi.org/10.1145/1294261.1294276

[76] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. 271–280. https://doi.org/10.1109/ISSRE.2012.22

[77] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A Taxonomy of Model-based Testing Approaches. *Softw. Test. Verif. Reliab.* 22, 5 (Aug. 2012), 297–312. https://doi.org/10.1002/stvr.456

[78] Eduard van der Bent, Jurriaan Hage, Joost Visser, and Georgios Gousios. 2018. 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)*. 164–174. https://doi.org/10.1109/SANER.2018.8330206

[79] Christopher Vendome, Daniel German, Massimiliano Penta, Gabriele Bavota, Mario Linares-Vásquez, and Denys Poshyvanyk. 2018. To Distribute or Not to Distribute? Why Licensing Bugs Matter. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 268–279. https://doi.org/10.1145/3180155.3180221

[80] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug Characteristics in Blockchain Systems: A Large-Scale Empirical Study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 413–424. https://doi.org/10.1109/MSR.2017.59

[81] Aaron Weiss, Arjun Guha, and Yuriy Brun. 2017. Tortoise: Interactive System Configuration Repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 625–636. http://dl.acm.org/citation.cfm?id=3155562.3155641

[82] Gerrit Wikimedia. 2019. Project operations/puppet/cdh. https://gerrit.wikimedia.org/r/#/admin/projects/operations/puppet/cdh. [Online; accessed 21-August-2019].

[83] Xin Xia, Xiaozhen Zhou, David Lo, and Xiaoqiong Zhao. 2013. An Empirical Study of Bugs in Software Build Systems. In *2013 13th International Conference on Quality Software*. 200–203. https://doi.org/10.1109/QSIC.2013.60

[84] Wei Zheng, Chen Feng, Tingting Yu, Xibing Yang, and Xiaoxue Wu. 2019. Towards understanding bugs in an open source cloud management stack: An empirical study of OpenStack software bugs. *Journal of Systems and Software* 151 (2019), 210 – 223. https://doi.org/10.1016/j.jss.2019.02.025

[85] Stefano Zilli. 2014. Hide secrets from puppet logs. https://bugs.launchpad.net/puppet-ceilometer/+bug/1328448. [Online; accessed 19-August-2019].

[86] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. 2018. How Practitioners Perceive Automated Bug Report Management Techniques. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2870414