As Code Testing: Characterizing Test Quality in Open Source Ansible Development

Mohammad Mehedi Hassan* Akond Rahman†
*Independent University, Dhaka, Bangladesh
†Department of Computer Science, Tennessee Tech University, Cookeville, TN, USA
Email: *mehedi.bueteee.23@gmail.com †arahman@tntech.edu

***Abstract*—Infrastructure as code (IaC) scripts, such as Ansible scripts, are used to provision computing infrastructure at scale. Existence of bugs in IaC test scripts, such as, configuration and security bugs, can be consequential for the provisioned computing infrastructure. A characterization study of bugs in IaC test scripts is the first step to understand the quality concerns that arise during testing of IaC scripts, and also provide recommendations for practitioners on quality assurance. We conduct an empirical study with 4,831 Ansible test scripts mined from 104 open source software (OSS) repositories where we quantify bug frequency, and categorize bugs in test scripts. We further categorize testing patterns, i.e., recurring coding patterns in test scripts, which also correlate with appearance of bugs. From our empirical study, we observe 1.8% of 4,831 Ansible test scripts to include a bug, and 45.2% of the 104 repositories to contain at least one test script that includes bugs. We identify 7 categories of bugs, which includes security bugs, and performance bugs that are related with metadata extraction. We also identify 3 testing patterns that correlate with appearance of bugs: 'assertion roulette', 'local only testing', and 'remote mystery guest'. We conclude our paper by discussing the implications of our findings for practitioners.**

***Index Terms*—ansible, bug, configuration as code, empirical study, infrastructure as code, quality, test**

I. Introduction

Infrastructure as code (IaC) scripts, such as Ansible scripts, are used to provision computing infrastructure at scale [33]. Use of IaC has yielded benefits for information technology (IT) organizations. For example, the use of Ansible scripts helped the U.S. National Aeronautics and Space Administration (NASA) to reduce its multi-day patching process to 45 minutes [23]. Swisscom, a Switzerland-based telecommunication provider, used Ansible to save 3,000 hours of IT administration work [5].

While IaC scripts are helpful for automated provisioning, prior research has reported these scripts to contain bugs [54], which can create large-scale consequences. For example, a buggy IaC script was the root cause of an outage that resulted in business losses worth of 150 million USD for Amazon Web Services (AWS) [30], [54]. Application of recommended software engineering practices, such as testing, can help practitioners mitigate bugs in IaC scripts [18]. Testing is beneficial for (i) gaining insights on the impact of changes made in IaC scripts, and (ii) building confidence in the IaC code base [59].

The 'as code' suffix in the phrase 'infrastructure as code' refers to the application of recommended software engineering practices, such as applying quality assurance activities for both development and testing for IaC scripts [33], [44]. Gaining an understanding of bugs is the first step towards adequately applying quality assurance activities for IaC test scripts, such as Ansible test scripts. Through a systematic characterization study we can quantify the frequency of bugs, as well as determine the categories of bugs that occur in Ansible test scripts. Such characterization can be helpful for researchers and practitioners in gaining an understanding of the nature of bugs in Ansible test scripts. Furthermore, such characterization study can also identify recurring coding patterns in test scripts, i.e., testing patterns, which also correlate with appearance of bugs. Identification of these testing patterns can guide practitioners on how to prioritize inspection efforts for bug identification, and adequately conduct Ansible testing—a need emphasized by industry practitioners, who reported a lack of guidelines on how to conduct IaC testing [27]. While characterization of bugs in testing artifacts for general purpose programming languages (GPLs) have been extensively investigated in prior work [8], [9], [66], such characterization remains under-explored for IaC languages, such as Ansible.

Anecdotal evidence from open source software (OSS) repositories further motivate us to characterize test quality in OSS Ansible development. Let us consider Figure 1, which presents code snippets downloaded from an OSS repository [1]. Figure 1a provides an example of a bug where '`/tmp/linchpin/bin/activate`' was provided as an incorrect configuration value [12]. The bug was fixed by providing the correct configuration value, which is '`/tmp/tutorial-env/bin/activate`'. Furthermore, as shown in Figure 1b, the bug occurred in a test script, which uses the coding pattern `hosts: localhost` to specify testing only in the local development environment. Conducting IaC testing only in the local development environment is limiting as test failures that occur in a remote cloud-based environment might not be manifested in a local development environment [11]. Appearance of a bug in a test script that contains the coding pattern `hosts: localhost`, suggests a relationship between occurrences of certain testing

[1] https://github.com/CentOS-PaaS-SIG/linchpin

```
    ...
    shell: |
-     source /tmp/linchpin/bin/activate
+     source /tmp/tutorial-env/bin/activate
      python setup.py test
    args:
      chdir: ../../
      ...
```

a

```
---
- name: Unit tests for Linchpin
  hosts: localhost #Testing in the local environment only
  gather_facts: False
  tasks:
    - name: debug
      debug:
        msg: "Enabling contra-hdsl tests"
    - name: shell for ansible version
      ...
```

b

Fig. 1: Anecdotal evidence of bugs in OSS Ansible test scripts. Figure 1a shows code changes to fix a bug for an Ansible test script. For the same test script, the coding pattern `hosts: localhost` appears indicating a relationship between bug and testing pattern occurrences.

patterns and occurrences of bugs. This relationship, however, is subject to empirical substantiation.

Accordingly, we answer the following research questions:

- ***RQ1****: How frequently do bugs appear in Ansible test scripts? What categories of bugs appear in Ansible test scripts?*
- ***RQ2****: What categories of testing patterns correlate with appearance of bugs for Ansible test scripts?*

We conduct an empirical study with Ansible test scripts collected from 104 OSS repositories. We apply qualitative analysis [58] with 2,606 commit messages used for 4,831 test scripts to quantify the frequency of bugs. With open coding [58], we identify bug categories for Ansible test scripts. Furthermore, we apply open coding and statistical analysis to identify testing patterns in OSS Ansible development that correlate with appearance of bugs. Source code and datasets used in the paper are available online [6].

Contributions: We list our contributions as the following:

- A list of bug categories that appear in Ansible test scripts (Section IV-A);
- An empirical evaluation of how frequently bugs appear in Ansible test scripts (Section IV-A); and
- A list of testing patterns that correlate with appearance of bugs in Ansible test scripts (Section IV-B); and
- A tool called **T**est **Pa**ttern **M**iner for **A**nsible (**TAMA**) that identified testing pattern instances that correlate with appearance of bugs in Ansible test scripts (Section IV-B).

We organize the rest of the paper as follows: we discuss background and related work in Section II. We provide the methodology of our empirical study in Section III. We report our findings in Section IV. We discuss our findings and limitations respectively, in Section V and VI. Finally, we conclude the paper in Section VII.

II. Background and Related Work

We provide background on testing in Ansible and discuss related work in this section.

A. Background

IaC scripts are used to provision computing infrastructure at scale [33]. Along with automation, the practice of IaC also recommends treating scripts as 'first class citizens', i.e., applying recommended software engineering practices, such as version control, testing, and linting. The practice of IaC is implemented using tools, such as Ansible, Chef, and Puppet. In recent years, the use of Ansible amongst IT organizations has gained popularity to implement IaC [2], [56].

We provide background on Ansible scripts as we use Ansible scripts to conduct our empirical study. Practitioners develop Ansible scripts using Yet Another Markup Language (YAML) syntax. An Ansible script contains a set of tasks also referred to as 'plays'. Listing 1 provides an example Ansible script that creates a file called 'sample.txt' in the '/tmp' directory using a play called 'Create sample.txt'. As shown in Listing 1, Ansible provides dedicated syntax, such as tags to execute provisioning tasks. For example, `name`, `file`, `path`, `state`, `owner`, and `group` are examples of tags.

```
#This is an example Ansible script
- name: Create sample.txt
  hosts: localhost
  file:
    path: /tmp/sample.txt
    state: touch
    owner: test
    group: test
  register: temp\_file
```

Listing 1: An example Ansible script used to create an empty file called 'sample.txt'.

Testing in Ansible is conducted using tags, such as `assert` to validate if executed tasks made expected changes to the provisioned environment. Listing 2 shows an example where the existence of the created file in Listing 1 can be tested using the `assert` tag using a play called '*test if /tmp/sample.txt*

exists '. Along with Ansible-provided tags, practitioners can use their own modules to conduct Ansible-related testing. A test script can include one or multiple test plays.

```
- name: test if /tmp/sample.txt exists
    msg: "Sample file exists"
  assert:
    that:
       - temp_file.stat.exists == True
```

Listing 2: Example test script that tests existence of a file.

B. Related Work

We organize this section by first discussing prior research that have focused on IaC. Second, we discuss prior work related to test quality. Finally, we discuss existing research on bug categorization.

1) Prior Research on IaC: Our paper is closely related to prior research on IaC scripts. In prior work researchers have focused on code maintainability aspects, e.g., Schwarz [60] and Sharma et al. [63], in separate research studies investigated code maintainability aspects of IaC scripts. Quality issues of IaC scripts have also garnered interest. Rahman et al. [54] constructed a defect taxonomy for IaC scripts that included eight defect categories. In another work, Rahman et al. in separate studies quantified security weaknesses that appear in Ansible [56], Chef [56], and Puppet scripts [55]. Testing issues in IaC scripts have also been investigated by researchers. Hanappi et al. [28] investigated how the convergence of IaC scripts can be automatically detected, and proposed an automated model-based detection framework for convergence. Ikeshita et al [37] proposed a test suite reduction technique for IaC scripts. Hummer et al. [34] observed that testing in IaC is different to that of GPLs, and created a tool [35] to automatically test the idempotency of IaC scripts. Shambaugh et al. [62] constructed a technique called Rehearsal that tests if Puppet scripts are deterministic.

2) Prior Research on Test Quality for GPL-based Projects: Our paper is related to prior research that have investigated test quality for project developed in GPLs, such as Java. Van Deursen et al. [67] identified 11 categories of test smells, i.e., testing patterns that can cause maintenance issues. Tufano et al. [65] built on top of van Deursen et al. [67]'s work to investigate how practitioners perceive 6 categories of JUnit test smells. Bavota et al. [8] reported test smells to negatively impact software maintenance. Spadini et al. [64] investigated JUnit test cases collected from 10 projects, and reported that test cases with test smells are more likely to be more defect-prone. Palomba et al. [52] proposed and reported TASTE to outperform code metrics-based techniques for test smell detection. Vahabzadeh et al. [66] identified five test bug categories by mining 5,556 OSS bug reports.

3) Research Related to Bug Categorization: Our paper is also related to prior research that has investigated bug categories for software systems. In 1992, Chillarege et al. [17] proposed the Orthogonal Defect Classification (ODC) technique that included eight bug categories. Categories proposed by Chillarege et al. [17] were used by Cinque et al. [19] to categorize bugs for air traffic control software. Later in 2008, Seaman et al. [61] extended ODC to derive 7 categories of requirements bugs, 10 categories of design and source code bugs, and 7 categories of test plan bugs.

Researchers have also constructed bug taxonomies in a bottom-up fashion for domain-specific software systems. For example, Humbatova et al. [32] mined GitHub issues and SO posts to derive a fault taxonomy for software projects that use deep learning. Rahman et al. [54] used open coding with commits to derive bug categories for Puppet scripts. In short, bug categorization has been an active research area, where researchers have focused on deriving bug categories for domain-specific software systems, such as deep learning and Puppet development.

In short, we observe a lack of research that systematically characterizes bugs in IaC test scripts. We address this research gap in our paper by quantifying bug frequency and identifying bug categories in Ansible test scripts. We also identify testing patterns that correlate with appearance of bugs.

III. Methodology

We describe our methodology in this section, which is summarized in Figure 2.

A. Answer to RQ1: How frequently do bugs appear in Ansible test scripts? What categories of bugs appear in Ansible test scripts?

We describe the methodology for answering RQ1 in the following subsections:

1) Dataset Collection: We collect repositories by mining OSS repositories hosted on GitHub. We use GitHub repositories as IT organizations tend to host their OSS projects on GitHub [1].

OSS repositories are susceptible to quality concerns, for example, repositories hosted on GitHub can be used for personal purposes that do not adequately reflect professional software development [45]. We apply the following criteria to collect necessary repositories: *Criterion-1*: The repository includes Ansible scripts. We apply this criterion by inspecting if repositories include YAML files and also include directories with the keyword 'playbook'. Ansible scripts are organized as playbooks and stored in directories that are labeled as 'playbook' [56]. *Criterion-2*: The repository is not a clone of another repository. *Criterion-3*: Count of developers is at least four. Using this criterion, we filter out repositories that are used for personal purposes, such as projects used for coursework. *Criterion-4*: The repository has at least two commits per month. Munaiah et al. [45] used the threshold of at least two commits per month to determine which repositories have enough development activity. *Criterion-5*: The lifetime of repository is at least one month. Using this criterion, we filter repositories that have a short lifetime. We measure lifetime of a repository by calculating the difference between the last commit date and the creation date for the repository. *Criterion-6*: Similar to prior work on IaC [54]–[56], the proportion

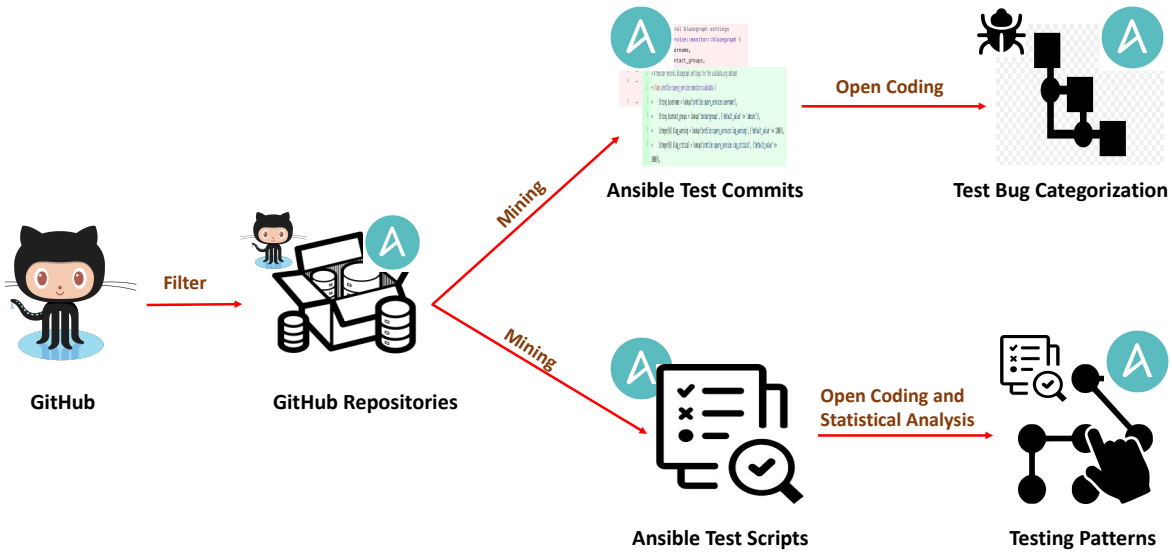

Fig. 2: An overview of our research methodology.

of Ansible scripts is at least 10%. IaC scripts can co-locate with other types of files, such as source code files and build files [38]. We assume that by using this threshold we can exclude repositories that do not have sufficient Ansible scripts for analysis. *Criterion-7*: The repository must include at least one Ansible test script developed in YAML. With this criterion we can include OSS repositories that contains Ansible scripts used for testing as well. We manually inspect each sub-directory within a repository to identify test scripts.

We use Google BigQuery [31] to download OSS repositories. Table I summarizes how many repositories are filtered using our criteria. We download 104 repositories by cloning the master branches on November 2020. The average count of Ansible test script per repository is 46.4 (min=1.0, median=3.0, max=921.0). The average count of test play is 4.9 per test script (min=1.0, median=3.0, max=104.0). Other attributes of the collected repositories are available in Table II.

TABLE I: Repository Filtering

	GitHub
Initial Count	3,405,303
Criterion-1 (Ansible Usage)	6,633
Criterion-2 (Not a Clone)	4,147
Criterion-3 (Contributor Count>3)	856
Criterion-4 (Commits/Month >=2)	770
Criterion-5 (Lifetime>1 month)	675
Criterion-6 (10% Ansible Scripts)	324
Criterion-7 (Existence of Ansible Test Scripts)	104

2) Ansible Test Bug Quantification and Categorization: We answer RQ1 by reporting the count of bugs and the bug categories for Ansible test scripts. The first author of the paper with 7.5 years of professional software development experience executes the following three steps:

Step#1-Bug Identification via Qualitative Analysis: The first author manually inspects each commit message for each

TABLE II: Dataset Attributes

Attribute	**GitHub**
Total Repositories	104
Avg. Repository Lifetime (Months)	43
Total Commits	700,696
Total Ansible Scripts	33,681
Total Ansible-related Commits	276,104
Total Ansible Test Scripts	4,831
Total Test Plays	23,841
Total Commits Modifying Ansible Test Scripts	2,606
Total Lines of Code (LOC) in Ansible Test Scripts	166,592

of the 2,606 commits that are used to modify any of the 4,831 Ansible test scripts downloaded from Section III-A1. While inspecting the commit messages the first author uses the following definition to determine if a commit is used to resolve a bug "*an imperfection in a program that needs to be corrected or replaced*" [36]. Upon completion of this step, the first author separates commit messages that are related to a bug. These commit messages are labeled as 'bug-related', and used in Step#2. Next, we manually analyzed the code changes for each bug-related commit to determine if a test script is modified in a bug-related commit. If we determine a test script to be modified in a bug-related commit, then we label that test script as 'bug-related'. A test script that is not modified in any bug-related commit is a 'neutral' test script.

We quantify the frequency of test bugs by using four metrics: (i) count of bug-related commits, (ii) bug proportion using Equation 1, (iii) bug density per 1,000 lines of code (LOC) using Equation 2, and (iv) proportion of bug-related test scripts using Equation 3.

$$\text{Bug Proportion} = \frac{\text{Total count of bug-related commits}}{\text{Total count of commits}} \quad (1)$$

$$\text{Bug Density } (x) = \frac{\text{Total count of bug-related commits}}{\text{Total LOC for all test scripts/1,000}} \quad (2)$$

$$\text{Proportion of Bug-related Scripts } (\%) = \frac{\text{Count of bug-related test scripts}}{\text{Total count of test scripts}} * 100\% \quad (3)$$

Step#2-Bug Categorization via Qualitative Analysis: For identifying bug categories in test scripts the first author applies open coding [58]. Open coding is a qualitative analysis technique to identify themes from unstructured text based on similarity analysis [58]. The first author applies open coding to identify categories by inspecting each of the bug-related commits identified from Step#1. In our categorization, a commit message can map to multiple bug categories.

We do not use existing bug categorization frameworks, such as ODC and Seaman et al. [61]'s work, as these categorization frameworks could be inadequate for Ansible test scripts. Prior research [10], [32] has reported pre-defined bug categorization frameworks to be inadequate for emerging programming languages and ecosystems, such as IaC.

Step#3-Rater Verification for Bug Categorization: The bug categories identified from Step#2 are susceptible to rater bias as these categories are identified by the first author alone. We mitigate this limitation by allocating another rater who participate voluntarily. The rater is a PhD student and not an author of the paper. We allocate 2,606 commit messages to the rater to perform closed coding [58]. The rater reads each commit message, and determines if the message maps to any of the identified bug categories. Next, we record a Cohen's Kappa [21] of 0.87 between the first author and the PhD student, indicating 'almost perfect' agreement [41].

B. Answer to RQ2: What categories of testing patterns correlate with appearance of bugs for Ansible test scripts?

We describe the methodology for RQ2 as follows:

1) Qualitative Analysis: We perform qualitative analysis by *first* identifying a sample with a sampling procedure called purposeful sampling [43], [51] similar to prior research [7]. The first author applies purposeful sampling to identify Ansible test scripts from which we will identify testing patterns, i.e, recurring coding patterns in test scripts that may correlate with bugs in test scripts. As part of conducting purposeful sampling, the first author focuses on variation intensity as discussed by Palinkas et al. [51], and conducts preliminary exploration to understand the variation in the content of the 4,831 Ansible test scripts. Upon completion of the preliminary exploration, the first authors identifies a purposeful sample of 500 randomly-selected test scripts that is representative of the variation documented in the set of 4,831 test scripts.

Second, we apply open coding [58] where we manually inspect test scripts to identify testing patterns that may correlate with appearance of bugs. We use two raters, the first and last author of the paper to conduct the open coding process. The first and last author respectively, has an experience of 3 years and 6 years in working with IaC. The first and last author respectively, took 600 and 372 hours to individually complete open coding for the 500 test scripts. Upon completion of open coding, we record disagreements: the first author identify 4 categories, and the last author identify 3 categories. The Cohen's Kappa [21] is 0.43, indicating 'moderate' agreement, according to Landis and Koch [41]. The first and second rater rater reached qualitative code saturation [29], i.e., did not identify any new categories respectively, after inspecting 132 and 201 test scripts. This shows the set of 500 randomly-selected scripts to be adequate for identifying testing patterns.

Both raters discussed their disagreements and provided reasons for their categorization. The first author identified 'linter strangler', which was missed by the last author. The first author provided examples of linter strangler, i.e., disabling lint checking for test scripts, and discussed how disabling lint checking can cause maintainability issues. The last author was convinced by the arguments of the first author, and agreed that linter strangler should be considered as a testing pattern. After resolving disagreements the Cohen's Kappa [21] is 1.0, as both raters agreed on all testing patterns identified using open coding.

2) Quantify Testing Pattern Frequency: We quantify testing pattern frequency by applying the following steps using an automated tool called **T**est **P**a**t**tern **M**iner for **A**nsible (**TAMA**):

Parsing: TAMA parses test scripts collected from Section III-A1 into key-value pairs.

Rule Matching: From the parsed content of the test scripts, TAMA applies rule matching to identify testing patterns. The rules needed to identify testing patterns are listed in Table IV. These rules are identified by abstracting code snippets for each pattern. The rules presented in Table IV leverage pattern matching similar to prior research [53], [56]. The string patterns used by each rule in Table IV is provided in Table V.

Rule Derivation Process: We leverage commonalities in coding patterns, and abstract such commonalities as rules to detect pattern instances. We provide an example in Table III to demonstrate our rule derivation process. In the 'Coding Pattern' column, we observe two coding patterns that are instances of linter strangler. In both coding patterns, `tags` are used to specify the coding pattern `skip_ansible_lint`. TAMA can parse both coding patterns as key value pairs, where `tags` is the key and `skip_ansible_lint` is the value. In both coding patterns we notice commonality in the key value pairs, which can be abstracted to a rule $isTag(k) \wedge isSkipLint(k.value)$. We repeat the same abstraction process for other patterns identified in Section III-B1.

Evaluation of TAMA: We evaluate TAMA by constructing an oracle dataset. We construct the oracle dataset using a rater who is not the author of the paper. The rater applies closed coding [58], where the rater manually examines each

TABLE III: An Example to Demonstrate How Code Snippets are Used to Determine the Rule for 'Linter Strangler'

Coding Pattern	Parsing Output of TAMA

`tags:`
`  -`
`skip_ansible_lint`
	<Key, 'tags', `skip_ansible_lint` >

`tags:`
`  - swap-format`
`  -`
`skip_ansible_lint`
	<Key, 'tags', `swap-format` >, <Key, 'tags', `skip_ansible_lint` >

test script and provides a label for the test script. We do not impose any time limit for the rater to conduct closed coding. We provide the rater a guidebook that includes the names, definitions, and examples of each pattern. The guidebook is available online [6].

We use 100 randomly-selected test scripts that are not used to perform open coding in Section III-B1 to construct our oracle dataset. We use precision and recall to measure TAMA's detection accuracy. Precision is the fraction of correctly identified pattern instances among the total identified pattern instances, as determined by TAMA. Recall is the fraction of correctly identified pattern instances that have been retrieved by TAMA.

The rater took 40 hours to conduct closed coding. After completion of closed coding we run TAMA on the oracle dataset. The oracle dataset included 176 testing pattern instances in 96 test scripts. No testing pattern was identified in 4 test scripts. The precision and recall of TAMA for each identified category is listed in Table VI. The 'Count' column provides the count of reported instances for each pattern. We observe TAMA's average precision and recall respectively, to be 0.93 and 0.98 across all four categories. From our evaluation using the oracle dataset, we observe TAMA to have a recall of $>= 0.95$ across all four categories, which gives us the confidence that TAMA will correctly identify testing pattern instances but can generate false positives.

TABLE IV: Rules Used by TAMA

Category	Rule
Assertion Roulette	$isAssert(k) \wedge length(k.value) > 1$
Linter Strangler	$isTag(k) \wedge isSkipLint(k.value)$
Local Only Testing	$isHost(k) \wedge isLocal(k.value)$
Remote Mystery Guest	$isPlay(k) \vee isInstall(k) \vee isURL(k.value)$

TABLE V: String Patterns Used to Execute Rules in Table IV.

Function	String Pattern
$isAssert()$	'assert'
$isHost()$	'hosts'
$isInstall()$	'install'
$isLocal()$	'localhost'
$isSkipLint()$	'skip_ansible_lint'
$isTag()$	'tags'
$isURL()$	'http://', 'https://'

TABLE VI: Evaluation of TAMA with Oracle Dataset

Pattern Name	Count	Precision	Recall
Assertion roulette	22	0.91	0.95
Linter Strangler	15	0.93	1.00
Local Only Testing	32	0.91	1.00
Remote Mystery Guest	107	0.96	0.98
Average		0.93	0.98

3) Statistical Analysis: Upon identifying the testing patterns, we apply statistical analysis to quantify if each of the identified testing patterns correlate with bug-related test scripts identified in Section III-A2. For correlation analysis we *first*, compute the count of each identified testing pattern from Section III-B1 for both: bug-related test scripts and neutral test scripts. *Second*, we apply statistical analysis. Through statistical analysis if we determine the count of testing patterns to be significantly larger for bug-related test scripts compared to that of neutral test scripts, then we conclude that testing pattern to exhibit a correlation with bugs in test scripts. We state the following null and alternate hypothesis for each of the identified testing pattern in Section III-B1 to perform statistical analysis:

- *Null*: count of testing pattern x is not higher for bug-related test scripts than neutral test scripts.
- *Alternate*: count of testing pattern x is higher for bug-related test scripts than neutral test scripts.

We use the Mann-Whitney U test [42] to accept or reject the null hypothesis as the test makes no assumption on the underlying distribution of the data. Following Cramer and Howitt's observations [22], for Mann-Whitney U test we determine the difference to be significant if $p < 0.01$.

4) Compute Metrics: Our statistical analysis from Section III-B3 will yield a set of testing patterns that correlate with appearance of bugs in test scripts. Using TAMA described in Section III-B2 we *first*, calculate the count of testing patterns for each test script in our dataset. *Second*, we compute two metrics: 'Test Pattern Density', and 'Test Script Proportion'. Test pattern density measures the frequency of testing pattern instances, for every 1,000 LOC. Test script proportion refers to the percentage of test scripts that include at least one instance of testing pattern. We use Equations 4 and 5 respectively, to quantify test pattern density and test script proportion.

$$\text{Test Pattern Density } (x) = \frac{\text{Total pattern instances for category } x}{\text{Total LOC for test scripts}/1,000} \quad (4)$$

$$\text{Test Script Proportion } (x) = \frac{\text{Test scripts with at least one instance of pattern } x}{\text{Total test scripts in dataset}} * 100\% \quad (5)$$

IV. Results

We provide answers to our research questions as follows:

A. *Answer to RQ1: How frequently do bugs appear in Ansible test scripts? What categories of bugs appear in Ansible test scripts?*

We answer RQ1 in this section.

1) Frequency of Bugs in Ansible Test Scripts: Altogether, we identify 2,606 commits to be used to modify 4,831 Ansible test scripts. Of the 2,606 commits, 147 commits (5.6%) are bug-related. Out of 104 repositories, 47 repositories (45.2%) included at least one bug-related test script. A breakdown of bug frequency is available in Table VII where we report bug density, bug proportion, and bug-related test script proportion.

TABLE VII: Answer to RQ1: Bug Density, Bug Proportion, and Bug-related Test Script Proportion

Category	Count
Bug Density (per KLOC)	0.88
Bug Proportion	5.6%
Bug-related Test Script Proportion	1.8 %

2) Bug Categories for Ansible Test Scripts: We identify 7 categories of bugs in Ansible test scripts, as summarized in Figure 3. We alphabetically describe each of these categories:

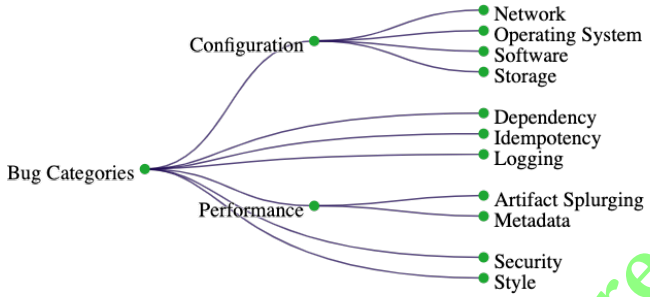

Fig. 3: Bug Categories for Ansible Test Scripts.

I. Configuration: Bugs that occur due to providing incorrect configurations. We identify four sub-categories: (i) network-related: bugs that are related with network devices and drivers; (ii) operating system (OS)-related: bugs that are related with configurations of operating systems; (iii) software-related: bugs that are related to incorrect configurations of a software package; and (iv) storage-related: bugs that are related with allocating and reusing storage utilities.

Example: We provide an example for each sub-category of configuration bugs: (i) network-related: a network misconfiguration in a test script occurred for using an unexpected Ethernet interface name [50]. The practitioner incorrectly assumed that only one public Ethernet interface name needs to be handled by the test script. The bug was fixed by adding a functionality to handle multiple public Ethernet interface names; (ii) OS-related: for an OSS project called 'os-ansible-deployment', network bootstrapping code in an Ansible test script was working correctly for Ubuntu, but not for CentOS. The bug occurred for not accounting the fact that the location of device and drivers varies from one OS to another [47]; (iii) software-related: a configuration called `MYSQLD_STARTUP_TIMEOUT` was not used in a test script, which resulted in a bug [49]. `MYSQLD_STARTUP_TIMEOUT` is a configuration related to the MySQL daemon `mysqld`; and (iv) storage-related: for an OSS project called 'ceph/ceph-ansible', an Ansible test was failing by allocating a 3 GB volume for storage, which was fixed by allocating 10 GB [13].

II. Dependency: Bugs that occur due to unavailability of a software artifact, which is necessary for a test script's execution.

Example: In the 'ceph/ceph-ansible' repository [14], execution of a test script failed because of a missing 'pyyaml' [2] dependency, which was not installed.

III. Idempotency: Bugs that occur when the idempotency property is violated for Ansible test scripts. Idempotency ensures a test script to yield the same results, when the same test script is executed multiple times.

Example: Idempotency bug was documented for an OSS repository called 'openstack/openstack-ansible', which lead to test case failures [46].

IV. Logging: Bugs that occur when inappropriate level of logging is used, e.g., duplicate logging, too little logging, or too much logging. Inappropriate level of logging can lead to debugging-related difficulties.

Example: For the 'ceph/ceph-installer' project, we observe an Ansible test script to include duplicate logging statements, which was deemed unhelpful for debugging [16].

V. Performance: Bugs that incur additional program execution time. We identify two sub-categories: (i) artifact splurging: bugs that occur due to not caching software artifacts, such as software packages and Docker images; and (ii) metadata: bugs that occur due to not using recommended Ansible code elements for extracting metadata of computing environments. One such code element is `ansible_facts` that provides an efficient approach to collect metadata about the computing environment, which is provisioned by the Ansible script [3].

Example: We provide examples of performance bugs by providing an example for each sub-category: (i) artifact splurging: For the 'kubernetes-sigs/kubespray' project, execution of Ansible test scripts was slow because of not caching Docker images [40]. Instead, the test script downloaded Docker images for every build. Downloading a Docker image from a Docker image repository, such as DockerHub [26] takes considerable amount of time depending on the size of image. The recommended approach is to cache Docker images in the environment where the test script is running, instead of downloading it repeatedly [25]; (ii) metadata: in the 'ceph/ceph-ansible' project, metadata from the computing environment was being extracted without using `ansible_facts`, which resulted in slow execution for the test script [15].

[2]https://pypi.org/project/PyYAML/

VI. Security: Bugs that violate any of the following security objectives: confidentiality, integrity, or availability [24].

Example: For an OSS project 'kubernetes-sigs/kubespray', the following security bugs were documented for a test script: (i) use of default token authentication, and (ii) transport layer security (TLS) being disabled for authentication [39]. These security bugs violate the security objective of integrity.

VII. Style: Bugs that occur for violating recommended coding style practices of Ansible. An Ansible test script can execute correctly but still can have style bugs that can cause maintainability issues.

Example: In an OSS project, a style-related bug was documented due to use of the `shell` module [48]. Use of `shell` is perceived by the Ansible community to create maintainability issues, and is discouraged for usage [3].

Frequency of Bug Categories in Ansible Test Scripts: Based on bug proportion and test script proportion values, configuration bugs is the most frequent. Considering bug density style-related bugs is the most frequent. We provide a complete breakdown in Table VIII. Of the 55 configuration bugs, 20.0%, 23.6%, 43.7%, and 12.7% are respectively, network, operating system, software, and storage bugs. Of the 17 performance bugs, 76.5% and 23.5% are respectively, related to artifact splurging and metadata extraction.

B. Answer to RQ2: What categories of testing patterns correlate with appearance of bugs for Ansible test scripts?

From our qualitative analysis we identify 4 testing patterns. As discussed in Section III-B, we use open coding as well as Mann Whitney U test to determine which testing patterns correlate with test-related bug scripts. While we identify four testing patterns with open coding, three of them show correlation with bug-related test scripts. As shown in Table IX, we observe a $p - value < 0.01$ for three testing patterns: assertion roulette, local only testing, and remote mystery.

1) Testing Patterns that Correlate with Appearance of Bugs in Ansible Test Scripts: We alphabetically describe the 3 testing patterns that correlate with appearance of bugs in test scripts as follows:

I. Assertion Roulette: The recurring coding pattern of testing more than one assertion using the `assert` tag. If one assertion fails, a practitioner may not be able to identify for which assertion the failure occurred. Instances of assertion roulette make troubleshooting of test failures harder. We report an instance of assertion roulette if more than one assertion is tested in one `assert` tag.

Example: In Listing 3 we observe a test play that tests three assertions using one `assert` tag: (i) if network interface is active (line#4), (ii) if interface type is 'bonding' (line#5), and (iii) and if maximum transmission unit (mtu) for the network interface is 9000 (line#6). Ansible will report '`fatal: FAILED`' message if any of the three assertions fail, or a combination of these three assertions fail. Troubleshooting the assertion failure can be challenging because to identify the location of the assertion failure, the practitioner has to comment one assertion at a time and rerun the test play.

```
1 - name: Bond check
2   assert:
3     that:
4       - ansible_bond0['active'] == true
5       - ansible_bond0['type'] == 'bonding'
6       - ansible_bond0['mtu'] == 9000
```

Listing 3: An example of assertion roulette where three assertions are tested.

II. Local Only Testing: The recurring coding pattern of only using the local environment to conduct Ansible testing. IaC testing can be conducted in local environments as well as remote environments [44]. However, while developing test scripts, practitioners may only execute tests in the local development environment, e.g., in their personal computer, and not in a remote environment, e.g., in an AWS instance. Test execution in local environments may not be reflective of remote cloud environments because system configurations, package dependencies, etc. in a local environment can be different to that of the remote cloud-based environments [33]. While testing in the local environment is better than doing no testing at all, local only testing is limiting, potentially leading to difficulty in debugging test failures. Hummer et al. [34] emphasized IaC testing to be conducted in remote environments because testing in remote environments can adequately ensure changes in the system state. We report an instance of local only testing when a test script uses the `hosts: localhost` coding pattern.

Example: We provide an example of using local only testing in Listing 4, where using the `hosts: localhost` coding pattern the practitioner specifies that creation of a network driver called 'networkd' will be tested in the local environment. If the test play is executed correctly then an active network bridge will be created. The test play can execute correctly in the local environment but not in a remote environment, which could have a different operating system and/or different system configurations. Behavior of device drivers, such as network device drivers are dependent on the type of operating system [57].

```
1 - name: Test networkd
2   hosts: localhost
3   connection: local
4   - name: Bridge check
5       assert:
6         that:
7           - ansible\_br\_dummy['type'] ==
              ↪  'bridge'
```

Listing 4: An example of local only testing: test execution is specified only for the local environment with the `hosts` tag.

III. Remote Mystery Guest: The recurring coding pattern of using a remote artifact for test play execution, which needs

TABLE VIII: Answer to RQ1: Frequency of Bug Categories in Ansible Test Scripts

Bug Category	Count	Bug Proportion (%)	Bug Density (per KLOC)	Script Proportion (%)
Configuration	55	2.11	0.33	0.72
Dependency	40	1.53	0.26	0.37
Idempotency	3	0.11	0.18	0.04
Performance	17	0.65	0.10	0.20
Logging	13	0.49	0.08	0.12
Security	11	0.42	0.07	0.16
Style	8	0.30	0.48	0.16

TABLE IX: $P - values$ for the Four Testing Patterns

Pattern Name	$P - value$
Assertion roulette	1.01×10^{-27}
Linter Strangler	0.20
Local Only Testing	4.08×10^{-09}
Remote Mystery Guest	7.24×10^{-08}

to be accessed via an HTTP/HTTPS-based URL. We count URL usage in a test script to quantify remote mystery guest instances.

Example: We use the code snippet presented in Listing 5 to demonstrate an example of remote mystery. Listing 5 presents a play titled 'Download LibVirt CPU map configuration script', which is dependent on the availability of a Python script called 'cpu_map_update.py'. Another practitioner can move the Python script of interest to another repository, which will make the test play fail. Furthermore, dependency on the remote Python script necessitates availability of network connectivity between the computing infrastructures, e.g., the infrastructure where the test script is executed, and the infrastructure where the Python script is hosted. The test play can fail if there is limited or no internet connectivity.

```
- name: Download LibVirt CPU map configuration script
  get_url:
    url: "http://git.openstack.org/cgit/openstack-dev/devstack/plain/tools/cpu_map_update.py?h=a631abadde7346b49"
    dest: /openstack/cpu_map_update.py
    validate_certs: yes
    mode: 755
  register: libvirt_cpu_map_download
  tags:
    - libvirt-cpu-map-download
- name: Test execution of LibVirt CPU map configuration script
  shell: /openstack/cpu_map_update.py /usr/share/libvirt/cpu_map.xml
  when: libvirt_cpu_map_download | changed
  tags:
    - libvirt-cpu-map-updated
```

Listing 5: An example of remote mystery guest where a remote Python file is imported using the `url` tag.

2) Frequency of Testing Patterns that Correlate with Appearance of Bugs: A breakdown of test pattern density and test script proportion is provided in Table X. The least and most frequent testing pattern that correlates with bugs is respectively, local only testing and remote mystery guest. In the case of assertion roulette, the median number of assertions per `assert` tag for GitHub is 2.0 (min = 2.0, max = 25.0). The count of unique Ansible test scripts in which at least one testing pattern appears is 957 out of 4,831 test scripts (19.8%).

TABLE X: Answer to RQ2: Test Pattern Density and Test Script Proportion (%) for the Three Testing Patterns

Pattern Name	Count	Test Pattern Density (per KLOC)	Test Script Proportion (%)
Assertion Roulette	527	3.16	7.15
Local Only Testing	245	1.47	4.71
Remote Mystery Guest	765	4.59	9.07

V. Discussion

We discuss the implications of our findings in this section.

Mitigation of Identified Bug Categories: IT organizations can mitigate the occurrence of bugs in Ansible test scripts by incorporating techniques and tools that target one or more of the identified bug categories. For example, static analysis tools such as SLAC [56] and 'ansible-lint' [4] can respectively, be helpful in mitigating security and style bugs. Mitigation of idempotency defects could be possible through early detection of idempotency with Hummer et al. [35]'s approach. The technique of interactive configuration repair proposed by Weiss et al. [68] can be used to repair configuration bugs.

Comparing Identified Bug Categories for Ansible Testing: Our identified bug categories have already been documented in existing research. For example, *configuration bugs* were also documented to appear for tests in Apache OSS projects and in OSS Puppet scripts. *Dependency, idempotency, and security* bugs have been reported for OSS Puppet scripts [54]. As documented in prior research, similar to Chef scripts [60] and Puppet scripts [63], style-related bugs also appear for Ansible test scripts.

Implications related to Prioritized Inspection: From Table VI we observe TAMA to have a precision and recall of > 0.90 for the 3 testing patterns: assertion roulette, local only testing, and remote mystery guest. Also, from Section IV-B2 we report that 19.8% of the 4,831 test scripts to include at least one instance of the 3 testing patterns. The implication of RQ2-related findings is that practitioners can use TAMA to automatically identify testing patterns that correlate with appearance of bugs. *First* with TAMA, practitioners can identify test scripts in which any of the 3 testing pattern appears. *Second*, they can

prioritize these test scripts for further inspection to identify bugs, as Table IX shows a relationship to exist between appearance of the 3 testing pattern categories and appearance of bugs. In this manner, practitioners can save inspection efforts for bug identification, as instead of inspecting all test scripts for bugs, they can inspect a smaller subset of test scripts.

Testing Patterns that Correlate With Appearance of Bugs - Similarities and Differences: From Section IV-B1 we identify three testing patterns that correlate with appearance of bugs. Of these three patterns assertion roulette and remote mystery guest have been documented as test smells in existing research [8], [65]. The implication of this finding is that the testing patterns that have negative implications for GPLs can also occur for non-GPLs, such as for Ansible test scripts.

We also notice one testing pattern to correlate with appearance of bugs that is unique to Ansible: local only testing. As Ansible is used to provision cloud computing resources, such as AWS instances, test results obtained by testing in the local environment may not generalize for remote environments, such as for cloud computing environments. Practitioners consider the activity of conducting IaC testing on remote environments as a good practice stating "*by running the tests on real systems, you can determine whether your application responded correctly in a realistic configuration*" [59].

Implications for Reproducible Deployments: One of the perceived benefits of IaC is reproducible deployments of cloud-based infrastructure, which enables practitioners to provision cloud-based infrastructure with consistent environments [33]. However, as shown in Section IV-B, practitioners use local only testing, which can undermine the value of IaC with respect to reproducible deployments. The example presented in Listing 4 tests functionality of network bridges only in the local development environment. The network bridge functionality may behave correctly for the local environment but not for one or multiple remote cloud instances due to differences in system configurations, package dependencies, etc., potentially creating inconsistencies between local and cloud-based environments. Local only testing is symptomatic of an 'uncontrollable configuration management process' [33], and is considered as a deterrent for reproducible deployments [11].

Implications for Troubleshooting Test Failures: From Section IV-B we observe test scripts can have as many as 25 assertions under a single `assert` tag. Existence of assertion roulette instances can negatively impact comprehension of test failures [8], which can make troubleshooting of test failures harder. Troubleshooting failures in cloud-based software development is challenging [20], and instances of assertion roulette can further aggravate the challenges that are related with cloud-based infrastructure maintenance.

Future Directions: Researchers can develop techniques that will investigate run-time behavior of test scripts and characterize potential flakiness in test scripts. Researchers can also investigate if other categories of testing patterns, which correlate with appearance of bugs, exist for Ansible scripts as well as for Chef, Puppet, and Terraform scripts.

VI. Threats to Validity

We discuss the limitations of our paper as follows:

Conclusion Validity: Our identified bug categories and testing patterns are limited to the dataset we used in Section III-A1. Also, the set of 500 Ansible test scripts used in Section III-B1 to determine testing patterns is subject to the first author's bias. The identified bug and testing pattern categories are susceptible to rater bias, which we mitigate by using two raters. We use commits to identify bug categories and bug-related test scripts, which can be limiting. Also, TAMA can generate false negatives and false positives when applied on other datasets. We mitigate this limitation by evaluating TAMA using an oracle dataset described in Section III-B2.

External Validity: Our datasets are constructed by mining OSS repositories. Our findings may not generalize for proprietary datasets. Also, our findings are limited to IaC scripts developed using Ansible, which may not generalize to other IaC languages, such as Chef and Puppet.

Internal Validity: While constructing the oracle dataset the rater may have expectations on the outcomes that could potentially impact the closed coding process. We mitigate the limitation by using a rater who is not an author of the paper. Furthermore, construction of the oracle dataset is susceptible to raters' experience in Ansible. We mitigate this limitation by providing the rater a document that describes each pattern with definitions and examples.

VII. Conclusion

The practice of IaC advocates for integrating quality into IaC development and testing. A characterization study of bugs in IaC test scripts, such as Ansible test scripts, is the first step towards aiding practitioners on how to integrate quality into IaC testing. Such characterization can also identify testing patterns that correlate with appearance of bugs in test scripts. We have conducted an empirical study with 4,831 Ansible test scripts mined from 104 OSS repositories. We observe bugs to appear in 1.8% of the 4,831 Ansible test scripts in our dataset. We identify 7 bug categories: configuration, dependency, idempotency, logging, performance, security, and style. We also identify 3 testing patterns that correlate with appearance of bugs namely, assertion roulette, local only testing, and remote mystery guest.

Based on our findings, we recommend application of techniques and tools that target one or more of our identified bug categories. We also recommend the use of TAMA to identify instances of the 3 testing patterns, as detection of testing patterns can help practitioners prioritize inspection efforts to find bugs in Ansible test scripts. We hope our paper will facilitate more research in the domain of IaC script quality.

ACKNOWLEDGMENT

We thank the PASER group at Tennessee Tech. University for their valuable feedback. The research was partially funded by the National Science Foundation award # 2026869.

REFERENCES

[1] A. Agrawal, A. Rahman, R. Krishna, A. Sobran, and T. Menzies, "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*, ser. ICSE-SEIP '18. New York, NY, USA: ACM, 2018, pp. 245–253. [Online]. Available: http://doi.acm.org/10.1145/3183519.3183549

[2] Alison DeNisco Rayome, "Ansible overtakes Chef and Puppet as the top cloud configuration management tool," https://www.techrepublic.com/article/ansible-overtakes-chef-and-puppet-as-the-top-cloud-configuration-management-tool/, 2019, [Online; accessed 25-Sep-2021].

[3] Ansible, "Ansible Documentation," https://docs.ansible.com/, 2021, [Online; accessed 19-Sep-2021].

[4] ——, "Ansible Lint Documentation," https://ansible-lint.readthedocs.io/en/latest/, 2021, [Online; accessed 29-Sep-2021].

[5] ——, "Swisscom Automates IT Management WITH RedHat Ansible Tower," https://www.ansible.com/hubfs/pdfs/RH-Ansible-Tower-swisscom-case-study.pdf?hsLang=en-us, 2021, [Online; accessed 13-Sep-2021].

[6] A. Authors, "Verifiability package for paper," https://figshare.com/s/4aa50ec7c34c18c71223, 2021, [Online; accessed 20-Sep-2021].

[7] T. Barik, D. Ford, E. Murphy-Hill, and C. Parnin, "How should compilers explain problems to developers?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 633–643.

[8] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 56–65.

[9] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.

[10] B. Beizer, *Software system testing and quality assurance*. Van Nostrand Reinhold Co., 1984.

[11] Y. Brikman, "5 lessons learned from writing over 300,000 lines of infrastructure code," https://blog.gruntwork.io/5-lessons-learned-from-writing-over-300-000-lines-of-infrastructure-code-36ba7fadeac1, 2018, [Online; accessed 14-Sep-2021].

[12] CentOS-PaaS-SIG/linchpin, "Update unit tests contra-hdsl," https://github.com/CentOS-PaaS-SIG/linchpin/commit/4905430ab36c, 2019, [Online; accessed 25-August-2021].

[13] ceph/ceph ansible, "tests: resize root partition when atomic host," https://github.com/ceph/ceph-ansible/commit/e1c1017e15, 2018, [Online; accessed 21-Jun-2021].

[14] ——, "ceph-ansible:Ansible playbooks to deploy Ceph, the distributed filesystem." https://github.com/ceph/ceph-ansible, 2021, [Online; accessed 23-Jun-2021].

[15] ——, "Use ansible_facts," https://github.com/ceph/ceph-ansible/commit/7ddbe747122, 2021, [Online; accessed 24-Jun-2021].

[16] ceph/ceph installer, "tests: remove duplicate logging statements," https://github.com/ceph/ceph-installer/commit/634cdc8b1f, 2017, [Online; accessed 24-Jun-2021].

[17] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943–956, Nov 1992.

[18] Chris Meyers, "Five Questions: Testing Ansible Playbooks & Roles," https://www.ansible.com/blog/five-questions-testing-ansible-playbooks-roles, 2017, [Online; accessed 22-Sep-2021].

[19] M. Cinque, D. Cotroneo, R. D. Corte, and A. Pecchia, "Assessing direct monitoring techniques to analyze failures of critical industrial systems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, Nov 2014, pp. 212–222.

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

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

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

[23] J. Davila, "Ansible/NASA Case Study," http://szsb-gl2x.accessdomain.com/fierce/wp-content/uploads/2016/01/NASA-Case-Study-Ansible.pdf, 2016, [Online; accessed 20-Jun-2021].

[24] G. Dhillon and J. Backhouse, "Current directions in is security research: towards socio-organizational perspectives," *Information systems journal*, vol. 11, no. 2, pp. 127–153, 2001.

[25] Docker, "Registry as a pull through cache," https://docs.docker.com/registry/recipes/mirror/, 2021, [Online; accessed 25-August-2021].

[26] DockerHub, "Build and Ship any Application Anywhere," https://hub.docker.com/, 2021, [Online; accessed 26-August-2021].

[27] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 580–589.

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

[29] M. M. Hennink, B. N. Kaiser, and V. C. Marconi, "Code saturation versus meaning saturation: how many interviews are enough?" *Qualitative health research*, vol. 27, no. 4, pp. 591–608, 2017.

[30] R. Hersher, "Incident documentation/20170118-Labs," https://www.npr.org/sections/thetwo-way/2017/03/03/518322734/amazon-and-the-150-million-typo, 2017, [Online; accessed 21-Sep-2021].

[31] F. Hoffa, "GitHub on BigQuery: Analyze all the open source code," https://cloud.google.com/blog/products/gcp/github-on-bigquery-analyze-all-the-open-source-code, 2016, [Online; accessed 16-Dec-2020].

[32] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1110–1121. [Online]. Available: https://doi.org/10.1145/3377811.3380395

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

[34] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Automated testing of chef automation scripts," in *Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference*, 2013, pp. 1–2.

[35] ——, "Testing idempotence for infrastructure as code," in *Middleware 2013*, D. Eyers and K. Schwan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 368–388.

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

[37] K. Ikeshita, F. Ishikawa, and S. Honiden, "Test suite reduction in idempotence testing of infrastructure as code," in *International Conference on Tests and Proofs*. Springer, 2017, pp. 98–115.

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

[39] kubernetes sigs/kubespray, "Security best practice fixes," https://github.com/kubernetes-sigs/kubespray/commit/d487b2f9279, 2017, [Online; accessed 24-Jun-2021].

[40] ——, "Refactor download role," https://github.com/kubernetes-sigs/kubespray/commit/66408a87ee, 2020, [Online; accessed 24-Jun-2021].

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

[42] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947. [Online]. Available: http://www.jstor.org/stable/2236101

[43] M. N. Marshall, "Sampling for qualitative research," *Family practice*, vol. 13, no. 6, pp. 522–526, 1996.

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

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

[46] openstack/openstack ansible, "Fix idempotency bug in AIO bootstrap," https://github.com/openstack/openstack-ansible/commit/a4dfb65169, 2016, [Online; accessed 24-Jun-2021].

[47] ——, "Use operating system specific IP utilities," https://github.com/openstack/openstack-ansible/commit/b697c55842, 2018, [Online; accessed 20-Jun-2021].

[48] openstack/openstack-ansible lxc_hosts, "Fix ansible-lint errors," https://github.com/openstack/openstack-ansible-lxc_hosts/commit/0d28eeab560, 2021, [Online; accessed 26-August-2021].

[49] os-cloud/openstack-ansible galera_server, "Updated repo for new org," https://github.com/os-cloud/openstack-ansible-galera_server/commit/cd11c5a56e96c0, 2015, [Online; accessed 27-August-2021].

[50] os-cloud/os-ansible deployment, "Fix main public interface name not always be eth0." https://github.com/os-cloud/os-ansible-deployment/commit/03d176d5a, 2016, [Online; accessed 22-Jun-2021].

[51] L. A. Palinkas, S. M. Horwitz, C. A. Green, J. P. Wisdom, N. Duan, and K. Hoagwood, "Purposeful sampling for qualitative data collection and analysis in mixed method implementation research," *Administration and policy in mental health and mental health services research*, vol. 42, no. 5, pp. 533–544, 2015.

[52] F. Palomba, A. Zaidman, and A. De Lucia, "Automatic test smell detection using information retrieval techniques," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 311–322.

[53] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "Tsdetect: An open source test smells detection tool," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1650–1654. [Online]. Available: https://doi.org/10.1145/3368089.3417921

[54] A. Rahman, E. Farhana, C. Parnin, and L. Williams, "Gang of eight: A defect taxonomy for infrastructure as code scripts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 752–764. [Online]. Available: https://doi.org/10.1145/3377811.3380409

[55] A. Rahman, C. Parnin, and L. Williams, "The seven sins: security smells in infrastructure as code scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 164–175.

[56] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3408897

[57] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser, "Dingo: Taming device drivers," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 275–288.

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

[59] D. Schmitt, "Hitchhiker's guide to testing infrastructure as/and code — don't panic!" https://puppet.com/blog/hitchhikers-guide-to-testing-infrastructure-as-and-code/, 2016, [Online; accessed 20-Jun-2021].

[60] J. Schwarz, A. Steffens, and H. Lichter, "Code smells in infrastructure as code," in *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2018, pp. 220–228.

[61] C. B. Seaman, F. Shull, M. Regardie, D. Elbert, R. L. Feldmann, Y. Guo, and S. Godfrey, "Defect categorization: Making use of a decade of widely varying historical data," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 149–157. [Online]. Available: https://doi.org/10.1145/1414004.1414030

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

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

[64] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 1–12.

[65] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 4–15. [Online]. Available: https://doi.org/10.1145/2970276.2970340

[66] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 101–110.

[67] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP)*, 2001, pp. 92–95.

[68] A. Weiss, A. Guha, and Y. Brun, "Tortoise: Interactive system configuration repair," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 625–636. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155641