It Works (only) on My Machine: A Study on Reproducibility Smells in Ansible Scripts

Ghazal Sobhani, Israat Haque, Tushar Sharma
Faculty of Computer Science
Dalhousie University
Halifax, Canada
{gh707451, israat, tushar}@dal.ca

***Abstract*—Infrastructure as Code (IaC) automates the creation, configuration, management, and monitoring of computing infrastructure through code. One of the key principles that IaC promises is repeatability and reproducibility. However, certain programming practices in IaC platforms, especially those that allow imperative configuration, such as Ansible, hinder reproducibility in IaC scripts. This study, first, identifies such programming practices that we refer to as reproducibility smells by conducting a comprehensive multi-vocal literature review and propose a first-ever validated catalog of reproducibility smells for IaC scripts. We implement a tool *viz.* REDUSE to identify reproducibility smells in Ansible scripts. Furthermore, we conduct an empirical study to reveal the proliferation of reproducibility smells in open-source projects and explore correlation and fine-grained co-occurrence relationships among them. We observe that *broken dependency chain* smell occurs the most in approximately 71% tasks that we analyzed. Our analysis uncovers significant positive correlations between specific reproducibility smells, implying that repositories with one such smell tend to exhibit others. Moreover, the co-occurrence analysis reveals smell pairs that show a high tendency of co-occurrence at the task granularity. With the developed tool REDUSE, DevOps engineers can identify and rectify reproducibility issues before becoming part of the production system. Software engineering researchers can use the smells catalog proposed first in this study and can utilize REDUSE in empirical studies exploring various facets of reproducibility.**

***Index Terms*—Infrastructure as Code, Ansible, Reproducibility, Reproducibility smells.**

I. INTRODUCTION

Infrastructure as Code (IaC) automates the creation, configuration, management, and monitoring of computing infrastructure through code, typically in the form of declarative configuration files [1], [2]. IaC offers many advantages over manual deployments, including faster, repeatable, and consistent deployment, improved scalability, enhanced reliability, and reduced operational costs [1].

Reproducibility is crucial in both scientific research and practical applications; it ensures the validity and reliability of findings and the feasibility of practices. Specifically, several attempts have been made to emphasize the importance of reproducibility in software engineering (SE) by revealing the deficiencies in current scientific practices [3]–[5]. In this paper, we focus on reproducibility in the context of IaC as it ensures the consistent and reliable provisioning and management of computing resources. Reproducibility in IaC refers to the ability to recreate an identical infrastructure environment using the same code and configuration files [6], [7]. The importance of reproducibility in IaC lies in its ability to guarantee consistency across different environments, such as development, testing, and production. By achieving reproducibility, organizations reduce the time to maintain computing infrastructure, improve productivity, and eliminate configuration drift [7], where inconsistencies and discrepancies arise between environments over time due to manual changes.

The quality of IaC scripts has been assessed in several studies. For example, Gonzalo *et al.* [5] provide recommendations for effective coding practices, including maintaining a consistent style, avoiding assumptions, organizing code predictably, and implementing error defense measures. Similarly, Kumara *et al.* [7], [8] emphasize the significance of reproducibility in IaC, particularly in maintaining consistent and easily reproducible environments. Rahman *et al.* [9] summarize approaches to identify defects in IaC scripts. While the research community has extensively examined practices impacting quality attributes such as security and maintainability in IaC [8]–[11], best practices for achieving reproducibility within IaC scripts and associated challenges remain largely unexplored in academic literature.

This paper addresses the research gap by first exploring existing knowledge about practices that affect reproducibility in IaC scripts in the existing literature in the form of a multi-vocal literature review. In this exploration, we choose to keep our focus on Ansible scripts (also known as playbooks) because, first, Ansible is one of the most commonly used IaC frameworks used for resource orchestration and configuration [8]. Also, the imperative nature of Ansible code, on the one hand, gives the flexibility to users to specify tasks utilizing known operating system commands but, on the other hand, poses a risk of leading to difficult-to-reproduce scripts (for example, due to violation of idempotency principle [8]). Our exploration led us to a set of practices hindering the reproducibility of Ansible scripts that we gleaned from academic articles and grey literature resources. We apply the open-coding method to analyze and synthesize a catalog of reproducibility smells, that we define as programming practice or specification that hinders the reproducibility of an IaC script, from the collection of practices. Then, we develop a tool REDUSE (REproDUcibility SmEll detector) to analyze Ansible playbooks and detect reproducibility smells. Furthermore, we conduct an empirical study to understand the proliferation of reproducibility smells and the degree of correlation and co-occurrence among them in open-source Ansible repositories. This paper makes following contributions to the field.

- The study introduces a comprehensive **reproducibility smell catalog** aggregating the existing knowledge from academic as well as grey literature for Ansible scripts. The catalog serves as a valuable resource for practitioners and researchers seeking to identify and mitigate reproducibility issues in Ansible scripts.
- The paper offers a tool REDUSE that we developed to **detect reproducibility smells**. Practitioners can use the tool to improve the reproducibility aspects of their Ansible scripts. Researchers in the field may use the tool to conduct studies to explore further ways to ensure reproducibility and investigate their causes and effects.
- Our **empirical study** explores properties of the reproducibility smells and their relationship with each other. Observations derived from the study improve our understanding of Ansible scripts, reproducibility smells, and their characteristics.
- We make our **replication package**, including the source code of the developed tool REDUSE, scripts we use to generate and analyze data, and results obtained from analyzing each open-source repository available online [12] for replication and extension.

II. RELATED WORK

A. Assuring IaC scripts quality

We divide the section into sub-categories of related work to ensure IaC script quality.

Best practices in IaC: We find studies investigating tools and techniques to ensure various quality aspects, including security and maintainability. Kumara *et al.* [7] reviewed the best practices of popular IaC languages, including implementation, design, and violations of IaC principles (*e.g.,* idempotence and separation of configuration code) useful to improve IaC development practices. Rahman *et al.* [9] curated approaches to identify defects in IaC scripts. Finally, Hummer *et al.* [13] focused on effective and efficient IaC configuration management by emphasizing automated configuration management and standardized workflows.

Code smells in IaC scripts: Several studies focus on evaluating the quality of IaC scripts and identifying code smells. Dallapalma *et al.* [10] propose a catalog of metrics to evaluate the quality of Ansible scripts and other IaC languages. Their work enables engineers to assess infrastructure code quality and support incremental refactoring efforts. Schwarz *et al.* [11] introduce 17 new code smells for Chef. The study compares the quality of IaC across different IaC frameworks. Sharma *et al.* [2] propose a catalog of implementation and design configuration smells for Puppet. Additionally, Rahman *et al.* [14] propose *AnsibleCheck*, a framework for automated detection of code smells in Ansible playbooks. Their work contributes to detecting and refactoring code smells, further enhancing the quality of Ansible scripts. These studies collectively contribute to evaluating and improving IaC script quality, helping practitioners identify and address code smells.

Security smells in IaC scripts: Several studies propose techniques for detecting security smells in IaC scripts. Dai *et al.* [15] developed *SecureCode*, an analysis framework that identifies risky patterns in infrastructure scripts and assesses their impact on business vulnerabilities, focusing on security issues. Opdebeeck *et al.* [16] develop a code smells detector for variables-related smells in Ansible scripts, highlighting the hazards related to variable reuse and overrides. Along similar lines, Opdebeeck *et al.* [17] propose an approach that leverages Program Dependence Graphs (PDG) to identify security smells in Ansible code. Bhuiyan *et al.* [18] focus on Insecure Coding Patterns (ICPs) in IaC scripts and propose a method to prioritize code review efforts, aiding practitioners in mitigating ICPs.

B. Reproducibility practices in software engineering

The reproducibility of scientific experiments is an overarching concern addressed in multiple studies. Some studies discuss general coding practices and toolkits that promote reproducibility in software development. For example, Gonzalo *et al.* [5] emphasize the importance of good coding practices, consistent coding style, documentation, and error handling. Their suggestions include utilizing version control, documenting code dependencies, simplifying execution, and utilizing containerization technologies like Docker. Feitelson *et al.* [19] propose approaches like comprehensive project documentation to improve the reproducibility of scientific experiments.

C. Comparison with the state of the art

While previous research in the IaC domain has focused on examining security and maintainability aspects within IaC scripts, our work focuses on the often-neglected yet vital concept of reproducibility. Incorporating reproducibility principles into IaC enhances the dependability and consistency of deployments. Our work first creates a comprehensive catalog of reproducibility smells by aggregating relevant information from multi-vocal resources. We develop a reproducibility smell detector for Ansible scripts. Using the developed tool REDUSE, we conduct an empirical study to explore the proliferation of reproducibility smells in open-source repositories and the relationship among smells to better understand reproducibility in the IaC domain.

III. STUDY DESIGN

The main *objective* of this research is to understand the difficulties involved in reproducing computing infrastructure in the context of IaC. We aim to gather challenges faced while reproducing IaC scripts and identify programming practices that lead to these challenges related to reproducibility. We refer the collection of such practices as *reproducibility smells*. Additionally, we aim to detect these reproducibility issues automatically to investigate the proliferation of such practices in open-source Ansible projects and understand them better by inferring relationships among the smells. We answer the following research questions (RQs) to achieve the above-stated goals.

RQ1. *What kind of programming practices impact reproducibility in IaC scripts?*

Identifying practices that impede reproducibility would guide developers to create scripts that consistently produce desired infrastructure setups across environments. With this research question, we want to consolidate the scattered knowledge about reproducibility in the IaC domain.

RQ2. *Which reproducibility smells are more prominent in open-source repositories?*

The research question aims to determine if certain reproducibility smells are more common than others in the analyzed open-source software repositories. The answer to this question can help developers be more cautious of reproducibility smells that are expected to occur more frequently and encourage them to take appropriate measures to address them.

RQ3. *Do reproducibility smells co-occur?*

A strong correlation between different types of smells can provide valuable insights into their occurrence patterns and inter-dependencies. Detecting how these smells cluster uncovers hidden complexities in IaC scripts, aiding developers in adopting holistic approaches to enhance script quality.

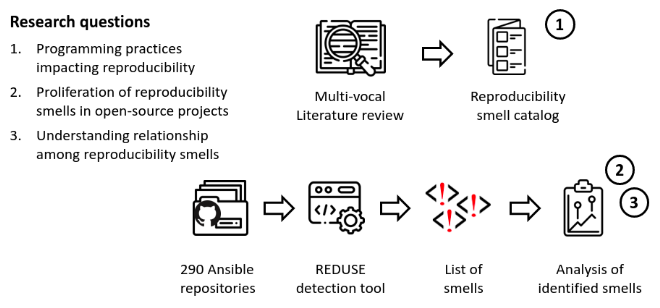

Research questions
1. Programming practices impacting reproducibility
2. Proliferation of reproducibility smells in open-source projects
3. Understanding relationship among reproducibility smells
Multi-vocal Literature review
Reproducibility smell catalog
1
2
3
290 Ansible repositories
REDUSE detection tool
List of smells
Analysis of identified smells

Fig. 1: Overview of the study.

We design our study consisting of multiple steps to answer the above research questions. Figure 1 provides an overview of the study. We carry out a multi-vocal literature review to search, filter, and consolidate programming practices affecting reproducibility to answer **RQ1**. The review helps in determining a catalog of reproducibility smells in IaC domain. Then, we implement a tool REDUSE to detect reproducibility smells in Ansible scripts. To investigate and address **RQ2** and **RQ3**, we identify a set of top open-source Ansible repositories from the Ansible Galaxy platform [20]. We use REDUSE to detect reproducibility smells in the identified Ansible repositories and discuss our observations based on the analysis. We also conduct a qualitative analysis to observe the impact of reproducibility smells in production IaC scripts.

IV. REPRODUCIBILITY SMELLS CATALOG

This section elaborates our approach to create a catalog of reproducibility smells and describes them using a template.

A. Catalog Creation

Given the practical nature of the problem space, we carry out a Multi-vocal Literature Review (MLR) [21]. An MLR combines information in academic research papers with grey literature sources, including blog posts, articles, programming discussion forums, and official documentation provided by framework or library developers. A wide range of resources helps us gain insights into the diverse range of efforts related to good and bad IaC practices. Through our extensive literature search and systematic cataloging effort, we contribute to the state of the art by thoroughly searching the academic and grey literature for reproducibility issues, systematically pruning and grouping these issues into distinct, actionable smells, and clearly defining and cataloging each smell. We elaborate on the adopted process in the rest of the section.

1) Resource gathering and searching: We follow practices employed in multi-vocal literature review [21]–[24] to search, filter, and identify resources for the review. Specifically, we use the Google search engine to search the grey literature, focusing on text-based sources such as reports, blog posts, white-papers, and official documentation related to IaC platforms. In the case of academic literature, we consider Google Scholar as well as IEEE and ACM digital libraries for literature search. We formulate the following search queries, inspired by similar studies [8], [25], [26], considering the scope of our search: *'Infrastructure as code' + smells + reproducibility*, *'Infrastructure as code' + anti-patterns + reproducibility*, *'Infrastructure as code' + bugs + reproducibility*, *'Ansible best practices' + reproducibility*, *'Ansible bad practices' + reproducibility*, and *'Ansible anti-patterns' + reproducibility*.

We carefully examine the search results corresponding to each search string and document the relevant resources (by reading the title of the resources) until we do not find new resources. In total we found 163 resources from grey sources after applying inclusion and exclusion criteria. Our replication package [12] includes the metadata of the resources (such as URL and title) along with the number of resources collected from individual search queries.

2) Inclusion and exclusion criteria: The inclusion criteria are as follows. First, the article must be written in English and have accessible full text. Also, the article should align with the focus of this study *i.e.,* covering practices to follow or avoid, discussing bugs, defects, smells, or anti-patterns, or describing challenges related to reproducibility in IaC, in general, or Ansible as a specific IaC automation framework.

We exclude resources that are duplicate or short. We also exclude articles that do not discuss reproducibility aspects, irrespective of whether IaC or not. Finally, articles that do not provide an adequate scope, rationale, consequences, or examples of recommended practices or practices to avoid such as bugs, defects, smells, and anti-patterns are also excluded.

We obtain a list of six academic and 110 grey literature resources after applying inclusion and exclusion criteria.

3) Snowballing: We employ both backward and forward recursive snowballing by carefully reviewing all the references and citations of our primary academic articles to identify relevant resources. For each potentially relevant article that we identify through snowballing, we then apply our inclusion and exclusion criteria. We add 26 articles from this exercise.

4) Quality assessment of grey literature: The nature of grey literature necessitates a careful evaluation to ensure the credibility and relevance of these sources. To this end, We follow the quality assessment best practices suggested in the literature [8], [26], [27].

We evaluate each grey resource on a scale of 20 points, covering *reputation of the publishing venue*, *author's expertise*, *clarity of its purpose*, and *publishing date*. Given that existing literature do not provide a systematic guidelines to evaluate and assign a score to grey resources beyond the above-mentioned evaluation criteria, we apply the following mechanism to thoroughly examine each resource.

- To assess **venue reputation**, a weighted approach considers affiliations (*i.e.,* institutional ties and collaborations), endorsements (*i.e.,* organizational support), publication history (including publication frequency and citation impact) of the venue, and reviews (in the form of user feedback). All of these parameters had the same weight.
- In evaluating **author expertise**, we consider academic credentials (*i.e.,* author's educational and professional background as well as specialization), publication track record, research affiliations, and professional experience.
- The evaluation of **content clarity** includes factors such as relevance, clarity, thorough analysis, novel insights, and illustrative examples.
- **Publication date** is used as a metric of recentness. Recently released resource scores higher than a relatively old resource.

The quality assessment process involves a thorough evaluation by two reviewers; both are graduate students with four years of experience in software development. One of the reviewers possesses expertise in the IaC field, whereas the second has a general acquaintance. Each reviewer independently examined the resources and assigned a score corresponding to each of the criterion mentioned above with five being the highest score and one is the lowest. All four aspects have the same weight in calculating the final score. After completing the exercise independently, we consolidate the scores. We obtain a high inter-rater agreement ($\kappa = 0.94$). If their individual scores for a specific aspect have a minor disagreement (*i.e.,* ± 1), the consolidated score is calculated by taking an average. However, when their individual scores have a substantial disparity, both reviewers engage in a discussion to understand the rationale for the provided scores and reach a consensus on the final score.

After evaluating all the resources, we discard resources with a score of less than ten in the quality assessment exercise. We also discard resources with a score less than three (out of five) in the *Relevance and clarity of content* aspect to keep the selected resources very relevant to our study, resulting in a total of 78 remaining resources. The marking guideline, the assessment document of each reviewer, and the detailed evaluation scores corresponding to each grey resource, including a summary, and the resource link, can be found in our replication package [12].

5) Data extraction and analysis: We combine resources obtained from both the academic and grey literature selection and filtering processes. We thoroughly study the combined list of resources and document their summary, key learning, and relevant metadata (*e.g.,* BibTex entry). We use the information summarized from the selected studies and resources to create a catalog of reproducibility smells.

We employed open coding [28], a qualitative method for analyzing and organizing concepts, to systematically categorize information from summaries of research papers and code examples. The lead author, with prior experience in qualitative research, conducted the initial coding phase. This involved thoroughly reviewing the collected information, breaking it down into smaller segments for detailed examination, and identifying relationships, similarities, and differences. Each segment received a descriptive label reflecting the main ideas or practices related to reproducibility. To ensure consistency and capture emerging themes, an iterative refinement approach was adopted. The lead author revisited segments and codes multiple times, and after each round of coding, other authors reviewed the codes and participated in further rounds of coding. This collaborative process continued until a consensus was reached on a final coding scheme that effectively captured the data. Employing emergent coding throughout, we ultimately identified and categorized six key concepts, which we then classified as reproducibility smells.

B. Results of RQ1—Reproducibility Smells Catalog

A *reproducibility smell* is a practice to specify, configure, or program IaC scripts that hinder the reproducibility of the script. Each smell represents a specific practice that violates the IaC principle of reproducibility and deviates from best practices, thereby compromising reproducibility. Considering that code smells inherently signal a potential issue rather than an actual problem [29], reproducibility smells similarly point to a violation of guidelines and a potential issue necessitating additional validation within the given context.

We describe each smell in our catalog using a name, description, example, and potential fixes, followed by rules to detection the smell automatically. To keep the paper succinct, we keep the code snippets for the discussed examples in our replication package [12]. Similarly, we include the most important references for each proposed reproducibility smell; an interested reader may find the full list of references in our replication package.

1) Broken dependency chain: A broken dependency chain occurs when a dependency or a required component, such as a package, cannot be installed or configured, preventing subsequent tasks of the script from executing successfully [7],

[13], [30], [31]. This disrupts system reproducibility by causing inconsistent environments across deployments.

Example: In an Ansible playbook [32] belonging to `ansible-jupyterhub-hpc` repository, a task `'install CHP proxy auth token'`, gets tokens from a source file, then uses these tokens to perform several operations. However, the task does not check the existence of the source file nor the correctness of the extracted tokens, which can lead to a broken dependency chain.

Potential fix: In this example, adding Ansible constructs `stat` and `when` that check the existence of the file and the correctness of the tokens can mitigate the issue [33].

Detection rule: We detect this smell in an Ansible task when the task installs a package using hard-coded keys or tokens, installs a package or library directly from an invalid URL, or uses a set of files without any checks to ensure the file's existence [30], [31], [34]. We do not detect the smell when the task use at least one of the checking constructs to ensure the correct task execution: `package-facts`, `debug`, `when`, `set-fact`, `assert`, `with-items` and `set-facts`. Having these checks ensures that prerequisites are verified before executing dependent tasks, preventing failures due to unmet dependencies.

2) *Outdated dependency:* This smell occurs when an Ansible script specifies an outdated version of a software library or package for installation [31], [34]–[38]. Outdated dependencies in Ansible scripts can expose the infrastructure to hidden bugs, regressions, and security vulnerabilities, ultimately hindering script reproducibility.

Example: An Ansible playbook [39] in repository `ansible-role-virtualenv` has a task named `'install virtualenv post packages'`. It installs the packages that are specified in a default list specified within the virtual environment package manager which could become outdated because no update policy is mentioned in the script and the version in the package manager is fixed.

Potential fix: Creating a requirement file with compatible versions is standard practice. Similarly, creating Docker images with all the necessary packages and libraries is also a typical practice. However, updating the dependencies and their corresponding stable versions is strongly recommended to avoid this smell. In the above example, we add a `state` Ansible property that only installs the missing packages with their latest stable and compatible version.

Detection rule: We detect this smell in an Ansible task when the task uses the package module to install a package, but it doesn't explicitly specify the `update` attribute with values such as `upgrade` or `upgrade-cache`. This omission may lead to the installation of outdated packages [37], [40]. We also detect this smell when the task does not perform any checks to verify if a newer version of the package is available *i.e.,* the task is missing `package_facts` or `check_mode` attributes.

3) *Incompatible version dependency:* This smell occurs when an Ansible playbook or role specifies a package or library version that is either no longer available in the target system's repositories, or, incompatible with other components in the configuration [31], [34], [35], [41]. Incompatible version dependencies in Ansible scripts can lead to missing packages, cascading dependency conflicts, and version resolution problems leading to challenges in reproducible computing environments.

Example: An Ansible playbook in repository `ansible-for-devops` [42] has a task namely `'Install Apache, MySQL, PHP, and other dependencies'`. The task installs a specific version of PHP extensions without verifying the absence of other versions in the environment. This oversight can lead to conflicts if another PHP version is already present. Specifying fixed package versions in this task can cause dependency conflicts, playbook failures due to unavailable versions, and cascading dependency issues.

Potential fix: Maintaining a compatible dependency matrix and setting the versions according to it may help avoid the smell. Often developers set `latest` to the `state` property to ensure the latest version of the package installation. While setting the state to `latest` might seem tempting, it can disrupt compatibility. A better approach might be to leverage tools such as `pipdeptree` to analyze existing dependencies and identify potential conflicts before introducing new versions. In addition, when installing any new dependencies using package managers, `upgrade_cache` should be used to ensure any previous version of the dependencies are removed from the environment.

Detection rule: We identify this code smell in Ansible tasks when the package module is used with a specific version number in the `version` attribute. We also detect this smell when the package module uses `state: latest`. This ensures up-to-date packages but may introduce unexpected changes and compatibility issues [40], [43].

4) *Assumptions about environment:* This smell arises when scripts rely on unverified assumptions about the environment, like the version or the type of the target machine's operating system or the availability of certain packages or libraries. General assumptions about the environment pose significant challenges in reproducing IaC scripts as they may not be accurate or applicable in all situations and environments [3], [5], [44], [45].

Example: In an Ansible playbook [46] in repository `chocolatey-ansible`, a task with the name `'checking if the bootstrap file has been created'` is using Windows-specific component without verifying the operating system of the environment.

Potential fix: Refactor scripts with parameters and conditions to adapt to different operating system versions and distributions using Ansible *facts* [33]. In the above example, we may use the `when` property to specify the required operating system as a condition to execute this task [33].

Detection rule: We detect execution environment-related assumptions and dependencies in Ansible, using various mechanisms, *e.g.,* checking for operating system-specific variables in a task, identifying commands tailored to a specific operating system, reviewing networking configurations (such DNS, firewall, and SSH), assessing assumptions about services and package repositories, examining path variables, and recognizing software-specific commands [44], [45], [47].

5) Hardware specific command: This smell occurs when scripts include commands or configurations that are tightly coupled to specific hardware components, such as particular CPU architectures or GPU models [37], [48]–[50]. Hardware-specific commands in Ansible scripts limit script portability and introduce hidden assumptions about the target hardware hindering script reproducibility across different hardware environments.

Example: A task `'Install AMD GPU drivers'` in repository `A:Platform64` [51] configures and installs AMD GPU drivers without checking the existence of this GPU in the machine.

Potential fix: Checking cross-device configuration compatibility by substituting hardware-specific commands with general tasks and using Ansible variables to abstract hardware-dependent values reduce the chances of failure and improves reproducibility. For the example presented above, we may add properties such as `when`, `debug`, `gather_facts` to the task to handle the situations when the expected GPU is not present in the target machine using the information gathered with variables in the task [33].

Detection rule: We detect *hardware specific command* smell when a hardware-specific command related to *disk management, system management, security management, performance and* GPU *settings, I/O management* components (such as `lspci`, `lshw`, `lsblk`, `fdisk`, and `parted`) is used in a task without confirming the presence of the hardware or an error handling code [48]–[50].

6) Unguarded operation: This smell occurs when Ansible tasks employ non-idempotent commands, particularly direct operating system commands, without appropriate state checks or safeguards. Such practices can lead to unintended system state changes and compromise the repeatability of Ansible playbooks [7], [34], [52], [53].

Example: A task `'Apply machine config'` in repository `openshift-ansible` [54] uses a command to perform an unguarded operating system-level operation without properly checking the status of the system.

Potential fix: Prioritizing idempotency in task design by using error handling constructs, such as `failed-when`, `changed-when`, and rigorously validating inputs and conditions before executing critical operations may avoid this smell. For the task in the example above, we may add a `changed-when` statement to ensure appropriate functionality. We can also use `when` to ensure the existence of the required configuration files and check if the configuration is not already applied.

Detection rule: To detect this smell, we identify tasks that execute unguarded operating system commands through Ansible that are not idempotent [13], [52], [53], [55], which include using package installers without checking the existence of the package state or version, manipulating files without idempotency checks and creating/updating *users*, *groups*, or *files* without idempotency considerations [30], [43], [56].

C. Smell catalog and detection rules validation

We identified eleven active IaC researchers and contacted them for feedback on the draft smell catalog and detection mechanism. Each question in the survey presented a reproducibility smell and corresponding detection mechanism. We asked the participants to rate the detection mechanism on a Likert scale ranging from one to five (the value five represents the detection mechanism is the most appropriate way to detect the smell). The survey was anonymous and open for ten days. One may find the survey questionnaire online [57].

We obtained a total of six responses. On average, the participants rated 3.33 for *broken dependency chain*, 2.5 for *outdated dependency*, 3.8 for *incompatible version dependency*, 4.0 for *assumptions about environment*, 4.0 for *hardware specific command*, and 3.8 for *unguarded operation*. These ratings show researchers' reasonable confidence in the proposed catalog and detection mechanism. *Outdated dependency* received the lowest confidence from the researchers. In fact, two of the researchers mentioned that the smell could not be considered as a reproducibility smell. Due to the overall low rating and the above comment, we decided to drop the *outdated dependency* smell from our catalog and the developed tool. In the rest of the paper, we will not include *outdated dependency* in our discussion.

Summary: After validating the catalog and detection rules obtained from our extensive multi-vocal literature review, we obtain a consolidated set of five reproducibility smells.

V. A Reproducibility Smell Detection tool

We developed Reduse (REproDUcibility SmEll detector)—a tool to detect reproducibility smells in Ansible scripts. Figure 2 shows the architecture of the tool.

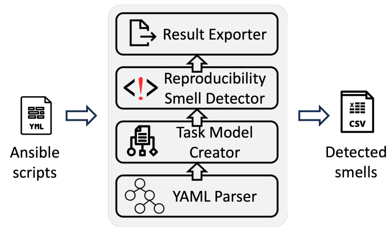


Fig. 2: Architecture of the smell detection tool.

We provide a path to the individual `yaml` file or a directory containing these files to invoke Reduse. It goes over each of the Ansible scripts individually and parses them using *PyYAML* [58] library. *PyYAML* parses the script and generates a nested list of key-value pairs containing identified Ansible constructs and their attributes in a structured manner.

Then, the *Task Model Creator* module uses the parsed script and generates key-value pairs to populate our custom source code model. It is a collection of *Task* instances; each *Task* instance represents an Ansible task and contains various task properties, including task name, target hosts, and Ansible-specific properties (*e.g., remote-user* and *gather-facts*). These properties hold essential information for the tool to detect reproducibility smells.

The *Reproducibility Smell Detector* module takes the task model instance for the Ansible script under analysis and detects reproducibility smells using the defined rules. Each smell detection rule requires checking specific properties of the specified Ansible tasks; if a rule is satisfied for a given task, we detect the smell in the task. Finally, the *Result Exporter* module emits the identified smells in a CSV file with details such as task name, file path, smell type, and a brief description, aiding developers in addressing potential issues.

A. Tool validation

We implement the smell detection rules discussed in the Section IV-B in our tool REDUSE. To assess the effectiveness of the developed tool, we conducted a manual validation that we elaborate on the next.

1) Data gathering: We first explore the search options in GitHub to identify Ansible repositories as our subject systems for validation; however, GitHub does not support searching repositories specific to frameworks. Moreover, given that `yaml` files can be used with other frameworks, such as Docker, a file-extension-based search approach also could not be used. To overcome the challenge, we decide to use Ansible Galaxy, as used by other studies in the domain [16], [25], [59]. First, we obtain the most downloaded repositories on the platform and then select repositories satisfying our selection criteria (*i.e.,* must have more than ten scripts, $1,000$ stars, and 300 commits) to avoid selecting small or low-quality repositories. We choose *ceph_ansible, openshift_ansible, ansible_for_devops* from this criteria. These repositories has 174 scripts and a total of $6,722$ Ansible tasks. We manually checked each repository to ensure that the repositories contain actual Ansible scripts. In addition, we also include the *oci-ansible-collection* [60] dataset, which has been used in testing GLITCH tool [25] and creating the Andromeda dataset [59]. This dataset contains 33 sub projects with a total 84 different Ansible scripts, containing $1,309$ individual Ansible tasks.

2) Evaluation approach: Two non-author evaluators, participated in the evaluation process. The evaluators possess knowledge of IaC concepts and Ansible; they are graduate students with approximately four years of software development experience. To minimize potential bias, we ensure to provide the evaluators with the definition, examples, and potential fixes of reproducibility smells without exposing them to the implementation of our tool. Both the evaluators independently assessed Ansible scripts and identified reproducibility smells. After the individual assessment was over, we matched findings from both evaluators and created a consolidated set of smells. The inter-rater agreement was high ($\kappa = 0.86$). In the case of different opinions, they discussed and resolved the differences. Once the ground-truth from the evaluators were established, we used our tool REDUSE to identify reproducibility smells in the selected repositories. The results from manual analysis and REDUSE are available in our online replication package.

TABLE I: Performance of REDUSE against manually annotated ground-truth

Smell	TP	FP	TN	FN	Precision	Recall	MCC	F1-score
Broken dependency chain	2,990	320	3,370	42	0.90	0.98	0.87	0.94
Incompatible version dependency	13	0	6,704	5	1.00	0.72	0.77	0.85
Assumptions about environment	2,092	780	3,710	140	0.73	0.94	0.65	0.82
Hardware specific command	10	0	6,712	0	1.00	1.00	1.00	1.00
Unguarded operation	2,780	298	2,340	10	0.93	0.96	0.89	0.94
Total	7,885	1,398	9,426	197	0.849	0.976	0.837	0.908

3) Evaluation: We compared the tool-generated results with manually curated ground-truth. Table I presents the results of the evaluation using typical metrics precision, recall, F1-score, and Matthews Correlation Coefficient (MCC). The tool performs well with F1-score = 0.908 and MCC = 0.837.

The high F1-score and MCC values indicate that the tool identifies reproducibility smells with a reasonable accuracy. The tool emits a few instances of false-positive for three smells. The tool, for example, incorrectly identifies *broken dependency chain* because it checks whether a task ensures its correct execution via specific Ansible attributes (*e.g., set-facts*, *package-facts*, and *assert*) within the task. However, there can be discrepancies when an Ansible script performs such checks in a different task. Since the tool currently does not support analysis across tasks, it reports false positive instances. Similarly, the tool could not identify a few instances of three smells. These false negative instances appear due to vast variations possible in Ansible specifications. For example, the tool implements a reasonable set of heuristics to check assumptions about the environment; however, there can be many other ways in which assumptions can be specified that the tool currently does not support. We aim to continue expanding the heuristics to benefit future developments in this field through our open-source repository.

VI. EMPIRICAL STUDY

A. Data collection

As we discuss in Section V-A, GitHub does not offer a convenient means to select a subset of Ansible repositories. We rely on Ansible Galaxy [20]—a hub for hosting, searching, and sharing Ansible projects for identifying our subject systems. Ansible Galaxy divides the hosted repositories into nine categories (*i.e., System, Networking, Database, Packaging, Security, Development, Cloud, Monitoring, and Web*). We apply selection criteria to identify the repositories for our empirical analysis. First, we select 100 most downloaded repositories from each category. The Ansible Galaxy platform does not provide a developer-friendly mechanism (*e.g.,* platform APIs similar to GitHub APIs) to extract the required information (in our case, a GitHub repository link as part of the metadata of the search Galaxy's result). To overcome the challenge, we developed a Python script to extract GitHub repository links from the web search results on Ansible Galaxy. We also use GitHub repository metadata to filter out low-quality and unmaintained repositories among the initial 900 repositories. Specifically, we select repositories with a minimum of 50 commits, at least five stars, and the last commit not older than a year. With these criteria in place, we obtained 290 repositories. We manually checked all these repositories looking for any repository that is example, test, or tutorial repository; we excluded such repositories. Our final dataset comprises 258 repositories; these repositories contain $4,100$ Ansible scripts and $19,412$ distinct Ansible tasks.

B. Results of RQ2

RQ2 aims to understand the proliferation of reproducibility smells in open-source repositories.

1) Approach: To answer this research question, we use REDUSE to identify the presence of reproducibility smells in all selected repositories. The tool identifies smells and stores the identified instances with relevant metadata (*i.e.,* repository name, script path, task name, the identified smells, and a brief description) for each analyzed Ansible script. We aggregate all the smell instances across all the analyzed repositories to calculate smell frequency across all Ansible tasks.

2) Results: Table II shows the total number of reproducibility smell instances detected in all the analyzed Ansible tasks. *Broken dependency chain* is the most frequently occurring smell. A high frequency of this smell indicates that software developers do not ensure the existence of required artifacts or dependencies in their infrastructure specifications. Similarly, *assumptions about environment* and *unguarded operation* smells show a high frequency, indicating that infrastructure specifications are written without concerning the portability of the instructions across various execution environments and without ensuring the idempotency properties of the specified operations. On the other hand, *incompatible version dependency* is the least occurring reproducibility smell (with only 15 instances), indicating that Ansible scripts typically do not mix package update policy for packages in a task. Finally, *hardware specific command*, with mere 42 smell instances, also rarely occur in Ansible scripts.

TABLE II: Frequency of detected reproducibility smells.

Reproducibility smell	Detected instances
Broken dependency chain	13,869
Incompatible version dependency	15
Assumptions about environment	5,511
Hardware specific command	42
Unguarded operation	5,295

Broken dependency chain exhibits the highest number of occurrences, approximately in 71% of Ansible tasks, among all the smells. The rationale of the high frequency can be traced back to the followed best practices and recommendations; according to them, the tool expects a defensive programming construct, such as `assert`, `when`, `stat`, or `debug` in each Ansible task. However, the majority of the Ansible tasks do not perform such a check and hence the tool reports a significantly high number of smell instances.

The identification of prevalent reproducibility issues in Ansible scripts offer crucial insights and actionable lessons. It underscores the necessity for meticulous dependency management, for script reliability. Additionally, it highlights the importance of script portability, urging developers to create robust, cross-platform scripts. The findings also signal a need for increased awareness and education about reproducibility best practices within the developer community, emphasizing the value of tooling and automation for proactive issue detection and resolution.

Summary: The prevalent *broken dependency chain* smell highlights a common oversight in ensuring necessary dependencies in Ansible scripts. Frequent occurrences of *assumptions about environment* and *unguarded operation* indicate problems with script portability and operating system-specific operations.

C. Results of RQ3

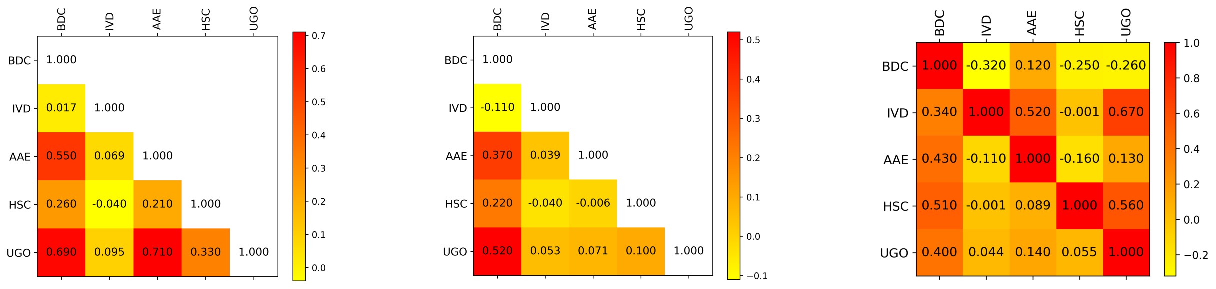
This research question investigates the relationships among reproducibility smells, specifically pair-wise correlation and co-occurrence. Analyzing how reproducibility smells co-occur in Ansible scripts may reveal underlying patterns and root causes, leading to a better understanding of these issues.

1) Approach: The process starts with a list of identified smells using REDUSE over all Ansible scripts. Then, we consolidate these smell instances at the repository granularity *i.e.,* each row represents the total number of smells per smell type for a repository. We use *Spearman* correlation analysis on each pair of smells to find correlation between all pairs of reproducibility smells.

To find out whether two reproducibility smells occur together in a task, we carry out a fine-grained analysis at the Ansible task granularity. We create a *contingency matrix* [61] for each pair of smells and compute ϕ co-efficient [62]. The ϕ coefficient provides a measure of co-occurrence, sensitive to both the presence and absence of reproducibility smells within individual tasks.

2) Results: First, we use Shapiro-Wilk test [63] to check the normality of the data distribution. We perform the test on each of the detected smells in all of the scripts; we obtain w in the range of 0.01–0.61 with the p-value < 0.05 for all the observations. This entails that the data is not following normal distribution. Therefore, we use Spearman correlation to measure the correlation between two reproducibility smells. Figure 3a presents the correlation coefficients between pairs of reproducibility smells; all the observations are statistically significant with p-values < 0.05.

We observe a high positive correlation between *unguarded operation* and *assumptions about environment* smells. The *unguarded operation* and *assumptions about environment* also exhibit high to moderate correlation with *broken dependency*



(a) Spearman correlation coefficients with absolute smell count.

(b) Spearman correlation coefficients with normalized smell count.

(c) Smell co-occurrence at the Ansible task granularity.

Fig. 3: Correlation and co-occurrence analysis. BDC refers to *broken dependency chain*, IVD to *incompatible version dependency*, AAE to *assumptions about environment*, HSC to *hardware specific command*, and UGO refers to *unguarded operation*.

chain smell. These high to moderate correlations among *unguarded operation*, *assumptions about environment*, and *broken dependency chain* suggest that if **a repository has a large number of one kind of smell among these three, it is likely to find other kinds of smells that show high correlation in the repository**.

The *number* of detected smell instances plays a role in the correlation analysis. Specifically, smells *unguarded operation*, *assumptions about environment*, and *broken dependency chain* show a high correlation with other types of smells as they are frequently detected. On the other hand, *hardware specific command* and *incompatible version dependency* smells that are the least frequently detected naturally show a low correlation with other smells. The size of the repository *i.e.,* the number of tasks in a repository may confound the analysis. To remove the factor of *size* from the analysis, we compute the normalized number of smells by dividing the total number of smells by the total number of Ansible tasks in a repository. We obtain a new set of correlation coefficients for the normalized smell count that we show in Figure 3b. The analysis shows an interesting observation. The erstwhile high correlation between, for example, *unguarded operation* and *assumptions about environment* smells is no longer visible. It implies that the high correlation was only due to the size of the repositories. With the normalized smell count, smell pair *unguarded operation* and *broken dependency chain* shows the highest correlation.

Furthermore, we investigate the *co-occurrence* relationship between pairs of reproducibility smells at the fine-grained task granularity. The *co-occurrence* relationship show whether two smells occur together at the *task granularity* whereas *correlation* capture tendency and proportion of smells to be detected for all the tasks in a *repository*. Figure 3c shows the calculated ϕ coefficient for each smell pair. The co-occurrence relationship is directional, unlike correlation, *i.e.,* co-occurrence between `(a,b)` is not equivalent to `(b,a)`.

We observe that the *unguarded operation* smell shows a high co-occurrence with *incompatible version dependency*. These smells usually occur when using package installers in an inappropriate way of specifying the package's version. Similarly, *assumptions about environment* smell exhibits a moderate degree of co-occurrence with *incompatible version dependency*. One potential reason for this relationship is that a script that assumes an execution environment may specify hard-coded versions for the required packages. The *hardware specific command* smell does not co-occur with other smells.

Summary: Our correlation analysis uncovers significant positive correlations between specific reproducibility smells, implying that repositories with one such smell tend to exhibit others. The co-occurrence analysis reveals the high to moderate co-occurrence tendency between specific smell pairs.

VII. Implications

A. Qualitative analysis

To further extend our exploration, we carry out a qualitative analysis. The goal of the analysis is to observe the manifestation of reproducibility smells in production IaC code belonging to open-source projects.

First, we identify the top 20 most downloaded Ansible repositories on Ansible Galaxy. We choose a subset of these repositories with at least 100 commits and at least ten reported issues on GitHub. We check each of the filtered project manually to ensure that these projects are real-world production-grade projects maintained by organizations such as Cisco or Redhat. Next, we conduct a thorough manual review of all the issues, closed and open, raised in the final set of eight repositories. In this review, we identify issues that originated during Ansible playbook execution mainly due to any of the reproducibility aspects discussed in this paper. During this analysis, upon finding such an issue, we also identify the potential reproducibility smells, capturing the reason for the issue. In other words, if the development team of the repository identified reproducibility smells, analyzed them, and refactored them early on, that would not have led to the issue.

In this exercise, we review 152 issues belonging to eight selected repositories. Among them, we identify 28 issues caused directly due to reproducibility aspects discussed in this paper. **Such a large proportion of issues arising due to reproducibility aspects clearly highlights the importance of analyzing Ansible scripts to find and refactor such issues early.** An interested reader may find the reviewed issues and corresponding mapping of reproducibility smells in our replication package [12].

For example, `cisco nxos` [64] repository manages and automates the NX-OS network appliances. We find two issues [65], [66] which are stemming from not using appropriate Ansible components when making changes in the files and not properly checking the state of the file after each change that led to idempotency violations which is detected as *unguarded operation* smell. Similarly, in another issue [67], the unguarded use of the `command` attribute without implementing status change measures resulted in a failure. This issue is captured by *unguarded operation*.

`Cisco asa` repository reports an issue [68] where a new version of a module was not compatible with another dependent module resulting in unexpected output in execution. It indicates *incompatible version dependency* smell that is caused by specifying a fixed version for a module without checking their compatibility. Another issue from the same repository [69] highlights a problem with an Ansible task that failed to verify its proper execution using appropriate Ansible components such as `assert`. This oversight led to errors in playbook execution, resulting in incorrect configuration of the environment based on the configuration file. Such issues can be captured using the *broken dependency chain* smell.

B. Implications

This study underscores the importance of adherence to the best practices in Ansible playbook development from reproducibility perspective. The study offers implications and actionable insights to software developers and DevOps engineers maintaining computing resources, researchers working in IaC domain, and tool vendors developing code analysis tools.

DevOps engineers can proactively identify and rectify reproducibility issues before they become integrated into production systems. Section VII-A highlights examples of reported issues that could have been prevented by early detection and correction of reproducibility smells. The study underscores the need for increased awareness and education within the developer community for proactive issue detection and resolution using automated tool support. Software engineering researchers can utilize and extend the smells catalog proposed in this study. They may also replicate and expand upon our empirical exploration using the provided tool, REDUSE. The discovery of correlated and co-occurring reproducibility smells presents an opportunity for holistic issue resolution, enabling developers and tool vendors to more effectively target interconnected issues. Furthermore, these correlations highlight the need for sophisticated tools capable of identifying and addressing interrelated problems. This insight can guide the enhancement of existing frameworks and methodologies in the field.

VIII. THREATS TO VALIDITY

Construct validity ensures the validity of constructs and abstractions used in a study. Reproducibility issues in Ansible playbooks are complex in nature, and our tool uses heuristics to identify the smells. Though the heuristics are grounded into semantics of IaC principles and guidelines, one may consider them as a threat to validity. We mitigated this threat by first validating the smells and corresponding rules by inviting elevan active researchers working in the field. Furthermore, we conducted tool validation by recruiting two evaluators with adequate knowledge of Ansible to validate the implementation. We have made the tool available online to promote critical review, replication, and extension [12].

A threat to *external validity* is that we only focused on Ansible as our target IaC framework; thus, it does not support other frameworks such as Puppet or Chef. However, our catalog of reproducibility smells is framework-agnostic and, thus, can be used with any imperative IaC framework.

Generalizability of literature and the risk of omitting relevant resources may threaten the external validity. We mitigate the risk by including a wider set of related terms that are not specific to Ansible but are applicable in general in the IaC context. We have also used incognito mode for our google search to avoid getting any biased results.

Internal validity ensures sufficient evidence to support the study's results and claims. We mitigate the threat by conducting a comprehensive literature search following the recommended practices for academic and grey literature. For the empirical study, we selected a set of repositories by specifying concrete quality criteria and identified 258 repositories belonging to diverse domains.

IX. CONCLUSIONS, LIMITATIONS, AND FUTURE WORK

By identifying and categorizing the issues that negatively affect reproducibility that we refer to as reproducibility smells, our research contributes to a systematic approach in modern computing infrastructure management. We introduce REDUSE that enables practitioners to detect these smells preemptively, mitigating potential impacts on production systems. Our empirical study targeted to explore prevalence of the reproducibility smells and relationships. To ensure transparency and facilitate further research, we have made all our code, scripts, and results publicly available [12].

Limitations: Code smells are, by definition, indicators of an issue (rather than an issue) [29], [70]. Reproducibility smells that we discuss in this paper are a type of code smell and hence share the same characteristics. Due to this, some of the identified smell instances can be false positives. The identified smells by REDUSE, or any smell detection tool based on heuristics and rules, need to be vetted considering the context. Furthermore, currently the granularity of analysis for the tool is an Ansible task. In the follow-up work, we intend to analyze multiple tasks simultaneously to capture any dependencies among them and produce a more accurate analysis.

Future work: Our future work involves exploring additional reproducibility smells in diverse project domains such as configuration projects for IoT devices. Furthermore, our detection tool can be refined by extracting and implementing additional rules considering developers' perspectives. We also aim to expand the scope of the study to other IaC tools and platforms such as Terraform, Chef, and Puppet.

References

[1] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*, 2016.

[2] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does Your Configuration Code Smell?" in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 189–200.

[3] C. Collberg and T. A. Proebsting, "Repeatability in Computer Systems Research," *Commun. ACM*, vol. 59, no. 3, p. 62–69, 2016.

[4] V. Stodden, J. Seiler, and Z. Ma, "An empirical analysis of journal policy effectiveness for computational reproducibility," *Proceedings of the National Academy of Sciences*, vol. 115, no. 11, pp. 2584–2589, 2018.

[5] G. Rivero and J. (Kristin) Chen, "Best coding practices to ensure reproducibility," 2020. [Online]. Available: https://griverorz.net/assets/pdf/good_practices.pdf

[6] K. Morris, *Infrastructure as code*. O'Reilly Media, 2020.

[7] I. Kumara, M. Garriga, A. U. Romeu, D. Di Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel, "The Do's and Don'ts of Infrastructure Code: A Systematic Gray Literature Review," *Information and Software Technology*, vol. 137, p. 106593, 2021.

[8] I. Kumara, Z. Vasileiou, G. Meditskos, D. A. Tamburri, W.-J. Van Den Heuvel, A. Karakostas, S. Vrochidis, and I. Kompatsiaris, "Towards Semantic Detection of Smells in Cloud Infrastructure Code," 2020.

[9] A. Rahman, E. Farhana, C. Parnin, and L. Williams, "Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts," pp. 752–764, 2020.

[10] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Toward a catalog of software quality metrics for infrastructure code," *Journal of Systems and Software*, vol. 170, p. 110726, 2020.

[11] 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.

[12] Anonymous, "REDUSE - Replication package," Aug. 2024. [Online]. Available: https://zenodo.org/records/13917319

[13] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Testing idempotence for infrastructure as code," in *Middleware 2013*, D. Eyers and K. Schwan, Eds., 2013, pp. 368–388.

[14] A. Rahman and L. Williams, "Source code properties of defective infrastructure as code scripts," *Information and Software Technology*, vol. 112, pp. 148–163, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584919300965

[15] T. Dai, A. Karve, G. Koper, and S. Zeng, "Automatically detecting risky scripts in infrastructure code," 2020, pp. 358–371.

[16] R. Opdebeeck, A. Zerouali, and C. De Roover, "Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime," 2022, pp. 61–72.

[17] R. Opdebeeck, A. Zerouali, and C. De Roover, "Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?" in *IEEE/ACM 20th International Conference on Mining Software Repositories (MSR 2023)*, 2023.

[18] F. A. Bhuiyan and A. Rahman, "Characterizing Co-located Insecure Coding Patterns in Infrastructure as Code Scripts," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 2020, pp. 27–32.

[19] D. G. Feitelson, "From Repeatability to Reproducibility and Corroboration," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, p. 3–11, 2015.

[20] "Ansible Galaxy," https://galaxy.ansible.com/home, 2023, last accessed: July 2024.

[21] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of systems and software*, vol. 138, pp. 52–81, 2018.

[22] C. Islam, M. A. Babar, and S. Nepal, "A multi-vocal review of security orchestration," *ACM Computing Surveys (CSUR)*, vol. 52, no. 2, pp. 1–45, 2019.

[23] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018.

[24] R. Verdecchia, I. Malavolta, and P. Lago, "Guidelines for architecting android apps: A mixed-method empirical study," in *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2019, pp. 141–150.

[25] N. Saavedra and J. a. F. Ferreira, "GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, 2023.

[26] M. Chiari, M. De Pascalis, and M. Pradella, "Static Analysis of Infrastructure as Code: a Survey," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, 2022, pp. 218–225.

[27] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, pp. 101–121, 2019.

[28] S. H. Khandkar, "Open coding," https://pages.cpsc.ucalgary.ca/~saul/wiki/uploads/CPSC681/open-coding.pdf, 2009.

[29] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018.

[30] Novaordis. (2021) Infrastructure as Code Concepts. Last accessed: July 2024. [Online]. Available: https://kb.novaordis.com/index.php/Infrastructure_as_Code_Concepts

[31] InfinityPP. (2020) Ansible Best Practices. Last accessed: July 2024. [Online]. Available: https://www.infinitypp.com/ansible/best-practices/

[32] "Exmaple—Broken dependency chain smell," https://gitlab.com/idris-cnrs/jupyter/ansible-jupyterhub-hpc/-/blob/main/roles/setup_jupyterhub/tasks/tokens.yml?ref_type=heads, 2023, last accessed: July 2024.

[33] Ansible, "Ansible Documentation," https://docs.ansible.com/ansible/latest/index.html, 2023, last accessed: July 2024.

[34] T. Das. (2022) Infrastructure as Code vs Configuration Management. Last accessed: July 2024. [Online]. Available: https://geekflare.com/infrastructure-as-code-vs-configuration-management/

[35] W. Network. (2020) Dependency Pinning with Infrastructure as Code. Last accessed: July 2024. [Online]. Available: https://wahlnetwork.com/2020/07/21/dependency-pinning-with-infrastructure-as-code/

[36] R. Wang, *Infrastructure as Code, Patterns and Practices: With examples in Python and Terraform*. Manning, 2022.

[37] R. Matsui. (2018) Stop Using Peer Dependencies. Last accessed: July 2024. [Online]. Available: https://medium.com/@arturkulig/stop-using-peerdependencies-e4b55755bd16

[38] R. Feintuch. (2018) New Security Challenges with Infrastructure as Code and Immutable Infrastructure. Last accessed: July 2024. [Online]. Available: https://thenewstack.io/new-security-challenges-with-infrastructure-as-code-and-immutable-infrastructure/

[39] "Example—Outdated dependency smell," https://github.com/cchurch/ansible-role-virtualenv/blob/master/tasks/update.yml, 2023, last accessed: July 2024.

[40] Thorntech. (2015) Infrastructure as Code Best Practices. Last accessed: July 2024. [Online]. Available: https://thorntech.com/infrastructure-as-code-best-practices/

[41] D. Enns. (2021) Troubleshooting Dependency Version Conflict. Last accessed: July 2024. [Online]. Available: https://github.com/autofac/Autofac/issues/1293

[42] "Example—incompatible version dependency smell," https://github.com/geerlingguy/ansible-for-devops/blob/master/includes/provisioning/tasks/common.yml, 2023, last accessed: July 2024.

[43] TechTarget. (2018) Perforce Acquires Puppet for Infrastructure as Code. Last accessed: July 2024. [Online]. Available: https://www.techtarget.com/searchitoperations/news/252515825/Perforce-acquires-Puppet-for-infrastructure-as-code

[44] Microsoft. (2023) Automation and Infrastructure. Last accessed: July 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/well-architected/devops/automation-infrastructure

[45] M. Ferrari. (2020) Want Repeatable Scale? Adopt Infrastructure as Code on GCP. Last accessed: July 2024. [Online]. Available: https://cloud.google.com/blog/products/devops-sre/want-repeatable-scale-adopt-infrastructure-as-code-on-gcp

[46] "Example—Assumption about environment smell," https://github.com/chocolatey/chocolatey-ansible/blob/9bdc0d40437a7dc7f0181af42da7e35bbcfcae4a/chocolatey/tests/integration/targets/win_chocolatey/tasks/bootstrap_tests.yml\#L4, 2023, last accessed: July 2024.

[47] S. IaC. (2023) Infrastructure as Code Security. Last accessed: July 2024. [Online]. Available: https://snyk.io/product/infrastructure-as-code-security/

[48] Snyk. (2023) Infrastructure as Code (IaC) - A Comprehensive Guide. Last accessed: July 2024. [Online]. Available: https://snyk.io/learn/infrastructure-as-code-iac/

[49] T. Guo. (2021) Managing Infrastructure with Terraform. Last accessed: July 2024. [Online]. Available: https://medium.com/geekculture/managing-infra-with-terraform-275968590fa4

[50] GitLab. (2022) Upgrading Auto Deploy Dependencies. [Online]. Available: https://docs.gitlab.com/ee/topics/autodevops/upgrading_auto_deploy_dependencies.html

[51] "Example—Hardware specific command smell," https://github.com/aplatform64/aplatform64/blob/3d563246263f6d2a83de604296704a4b76164caa/docs/examples/hw_gpu_amd.yml\#L4, 2023, last accessed: July 2024.

[52] O. Software. (2018) Organizing Ansible. Last accessed: July 2024. [Online]. Available: https://oteemo.com/organizing-ansible/

[53] Ansible. (2018) Ansible Best Practices Guide. Last accessed: July 2024. [Online]. Available: https://ansibledaily.com/idempotent-shell-command-in-ansible/

[54] "Example—Unguarded operation smell," https://github.com/openshift/openshift-ansible/blob/master/roles/openshift_node/tasks/apply_machine_config.yml, 2023, last accessed: July 2024.

[55] Ansible, "Ansible documentation—ad-hoc commands," 2024. [Online]. Available: https://docs.ansible.com/ansible/latest/command_guide/intro_adhoc.html

[56] S. BHARTIYA, "Importance Of Repeatability In IaC | Scaling Infrastructure as Code," https://tfir.io/importance-of-repeatability-in-iac-scaling-infrastructure-as-code/, 2022.

[57] "Online Questionnaire for IaC experts," https://forms.office.com/r/s27JTNDtrC, 2024.

[58] "YAML parser and emitter for Python," https://pypi.org/project/PyYAML/, 2024.

[59] R. Opdebeeck, A. Zerouali, and C. De Roover, "Andromeda: A Dataset of Ansible Galaxy Roles and Their Evolution," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 580–584.

[60] "Oracle Cloud Infrastructure Ansible Collection," https://github.com/oracle/oci-ansible-collection, 2023.

[61] S. Tsumoto and S. Hirano, "Contingency matrix theory," in *2007 IEEE International Conference on Systems, Man and Cybernetics*, 2007, pp. 3778–3783.

[62] Wikipedia contributors, "Phi coefficient — Wikipedia, the free encyclopedia," 2024. [Online]. Available: https://en.wikipedia.org/wiki/Phi_coefficient

[63] S. S. SHAPIRO and M. B. WILK, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3-4, pp. 591–611, 1965.

[64] "Cisco NXOS," https://github.com/ansible-collections/cisco.nxos, 2024.

[65] "Cisco NXOS," https://github.com/ansible-collections/cisco.nxos/issues/801, 2024.

[66] "Cisco NXOS," https://github.com/ansible-collections/cisco.nxos/issues/803, 2024.

[67] "Cisco NXOS," https://github.com/ansible-collections/cisco.nxos/issues/542, 2024.

[68] "Cisco asa," https://github.com/ansible-collections/cisco.asa/issues/195, 2024.

[69] "Cisco asa," https://github.com/ansible-collections/cisco.asa/issues/196, 2024.

[70] M. Fowler, P. Becker, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, 1999.