Do Experts Agree About Smelly Infrastructure?

Sogol Masoumzadeh, Nuno Saavedra, Rungroj Maipradit*, Lili Wei, *Member, IEEE*, João F. Ferreira, Dániel Varró, *Senior Member, IEEE*, and Shane McIntosh, *Senior Member, IEEE*

***Abstract*—Code smells are anti-patterns that violate code understandability, re-usability, changeability, and maintainability. It is important to identify code smells and locate them in the code. For this purpose, automated detection of code smells is a sought-after feature for development tools; however, the design and evaluation of such tools depends on the quality of oracle datasets. The typical approach for creating an oracle dataset involves multiple developers independently inspecting and annotating code examples for their existing code smells. Since multiple inspectors cast votes about each code example, it is possible for the inspectors to disagree about the presence of smells. Such disagreements introduce ambiguity into how smells should be interpreted. Prior work has studied developer perceptions of code smells in traditional source code; however, smells in Infrastructure-as-Code (IaC) have not been investigated. To understand the real-world impact of disagreements among developers and their perceptions of IaC code smells, we conduct an empirical study on the oracle dataset of GLITCH—a state-of-the-art detection tool for security code smells in IaC. We analyze GLITCH's oracle dataset for code smell issues, their types, and individual annotations of the inspectors. Furthermore, we investigate possible confounding factors associated with the incidences of developer misaligned perceptions of IaC code smells. Finally, we triangulate developer perceptions of code smells in traditional source code with our results on IaC. Our study reveals that unlike developer perceptions of smells in traditional source code, their perceptions of smells in IaC are more substantially impacted by subjective interpretation of smell types and their co-occurrence relationships. For instance, the interpretation of admins by default, empty passwords, and hard-coded secrets varies considerably among raters and are more susceptible to misidentification than other IaC code smells. Consequently, the manual identification of IaC code smells involves annotation disagreements among developers—46.3% of studied IaC code smell incidences have at least one dissenting vote among three inspectors. Meanwhile, only 1.6% of code smell incidences in traditional source code are affected by inspector bias stemming from these disagreements. Hence, relying solely on the majority voting, would not fully represent the breadth of interpretation of the IaC under scrutiny.**

***Index Terms*—Code Smells, Infrastructure as Code, Developer Perceptions, Oracle Datasets.**

S. Masoumzadeh and L. Wei are with the Department of Electrical and Computer Engineering, McGill University, Canada.
E-mail: sogol.masoumzadeh@mail.mcgill.ca, lili.wei@mcgill.ca
N. Saavedra and J. F. Ferreira are with INESC-ID and IST, University of Lisbon, Portugal.
E-mail: nuno.saavedra@tecnico.ulisboa.pt, joao@joaoff.com
R. Maipradit and S. McIntosh are with the David R. Cheriton School of Computer Science, University of Waterloo, Canada.
E-mail: rungroj.maipradit@uwaterloo.ca, shane.mcintosh@uwaterloo.ca
D. Varró is with the Department of Computer and Information Science, Linköping University, Sweden.
E-mail: daniel.varro@liu.se
*Corresponding author.

I. Introduction

Code smells are anti-patterns of code that indicate potential design or implementation setbacks in software code bases [1], [2]. Code smells violate code understandability, re-usability, and changeability, and increase the cost of code maintenance [3]. Hence, it is important to identify and locate code smells. To efficiently detect code smells, various automatic techniques are proposed [4]–[10]. The performance of these techniques are primarily assessed by manually constructing *oracle datasets* of code smell issues.

The oracle datasets are constructed by multiple developers independently annotating whether a set of code examples are prone to a list of smell types. If inspectors unanimously agree on the identified code smell type and the line of code where it is present, that line of code is annotated with the identified smell type as its *ground truth* label and is included as an issue in the oracle dataset. Since multiple inspectors cast votes about each code example, it is possible for the inspectors to disagree about existing types of code smells. In the case of disagreement, the *majority voting rule* is applied—if the majority of the inspectors identify the same code smell type at the same line of code, that line of code is annotated with that specific smell type as its ground truth. While the majority voting rule can be applied to discretize code examples (i.e., code is either smelly or it is not), it also conflates cases with unanimous agreements with cases that have some degree of disagreement or ambiguity. Cases with disagreements among inspectors raise doubt about ground truth labels of oracle datasets and even can degrade the performance of smell detection tools. Due to the negative impact of this ambiguity, prior work has studied the developer perceptions of code smells in traditional source code [11]–[13]; however, developer perceptions of code smells in other code-adjacent artifacts, such as *Infrastructure-as-Code (IaC)*, remains unexplored.

IaC is an emerging technique that is widely applied for automating the infrastructure to enable repeatable, scalable, and reproducible provisioning and imaging for systems with varied configuration [14]. As such, DevOps engineers can effectively manage the infrastructure using software development and release tools such as version control systems and continuous integration and deployment pipelines [14]. Due to its extensibility and reproducibility, incidences of code smells in IaC can have prominent negative consequences [15]. For instance, Listing 1 provides a real-world code smell incident in the Puppet code of *OpenStack*.[1] This example defines a notification transport layer and sets its attributes. The password of the transport layer is set as a hard-coded string on line

[1]The code example is accessible at https://is.gd/EXloIs.

Listing 1: Part of a real-world OpenStack Puppet code example

```
class { 'designate':
  default_transport_url   => os_transport_url({
   'transport' => 'rabbit',
   'host' => $::openstack_integration::config::host,
   'port' => $::openstack_integration::config::
       rabbit_port,
   'username' => 'designate',
   'password' => 'an_even_bigger_secret', })
```

7. This is dangerous, as the hard-coded password can leak, leading to a breach of sensitive data. In a similar scenario, when public-read IaC scripts of Amazon Web Services S3 billing system are tainted with hard-coded passwords, attackers can exploit them, causing a breach of end-user personal information and catastrophic financial losses for the service provider [16], [17]. Hence, in the current study we are set to be the first to investigate oracle datasets of IaC code smells for their possible subjectiveness and the implications these ambiguities may have for practitioners and researchers.

For this purpose, we conduct an empirical study and assess the extent to which oracle datasets of IaC code smells are impacted by ambiguity due to inspector disagreements. Specifically, we analyze the oracle dataset of GLITCH [15], a state-of-the-art detection tool for IaC *security code smells*—a particular family of smells which indicate the possibility for security breaches [18]. GLITCH is the only automatic detection tool for IaC code smells that provide complete access to its oracle dataset. Furthermore, as discussed by Guerriero et al. [19], IaC relies on heterogeneous tools with different natures and code models contributing to various challenges. Hence, the unique ability of GLITCH to detect various code smell types in multiple IaC technologies, compared to other detection tools such as Puppeteer [20] which only focuses on detecting specific code smell types in limited technologies, is an advantage for conducting the current study.

In our study, we also investigate whether confounding factors, such as annotation tendencies of individual inspectors or the types of code smells, are associated with the incidences of the misalignment of developer perceptions of IaC code smells. We structure our empirical study by addressing the following research questions.

RQ1: How prevalent are labeling disagreements?

Motivation: Labeling disagreements can introduce uncertainties and potential biases into the oracle dataset, consequently degrading the detection performance of the tool. Hence, we set out to understand to what extent developers disagree about the presence and type of IaC code smells.

Results: Nearly half of the oracle issues (i.e., 43.4%–48.7%) have some degree of disagreement among inspectors, suggesting that the oracle dataset may be impacted by the conflation of unanimous and ambiguous incidences of code smells.

RQ2: Do false alarms correlate with the annotations of the oracle dataset?

Motivation: False alarms can substantially impact the effectiveness of code smell detection tools, leading to distractions for developers and eroding their confidence in the tools. Annotation disagreements in the oracle datasets introduce an ambiguity that is not reflected in the perceived performances of the tools. Thus, in this research question, we set out to study the extent to which the high rates of disagreements that we observe in RQ1 impact the false alarms of GLITCH.

Results: In 53.6% of false alarms, the tool is entirely incorrect—it identifies a code smell incident while inspectors unanimously agree there exists no smell at the target line of code. The remaining 46.4% of the false alarms are raised in code for which at least one inspector raised a concern—either there exists an oracle issue of a different smell type (i.e., 25.7% of false alarms) or no two inspectors agree on the presence or the type of a smell incident—20.7% of false alarms.

RQ3: How labeling disagreements distribute among inspectors?

Motivation: It is possible that labeling disagreements are simply an artifact of a rogue inspector who always has a different perception of IaC code smells from the rest. For example, if one inspector incorrectly labels all instances of all smells, this would inflate disagreement rates without indicating anything of value about the underlying smell phenomenon (apart from the need for better inspector training). To study whether inspector tendencies explain labeling disagreements, in this research question, we study how disagreements are distributed among the inspectors.

Results: The distributions of labeling disagreements among the inspectors range from 24.5% to 38.8%, 15.8% to 47.4%, and 17.4% to 59.4% for Ansible, Puppet, and Chef IaC settings, respectively. The results demonstrate that there is no particular inspector who is disproportionately responsible for disagreements about IaC code smells.

RQ4: How are labeling disagreements distributed across the types of IaC code smells?

Motivation: It is also possible that labeling disagreements are simply due to difficulty of labeling a particular smell type. For example, if all of the disagreements occur because of one type of smell, this would inflate disagreement rates for smells while really being a localized problem with respect to that type. To study whether type tendencies explain misaligned perceptions, in this research question, we study how disagreements are distributed among the studied types of IaC code smells.

Results: No single type of IaC code smell dominates, with the most frequent types accounting for 1.6% to 35.4% of the total disagreements; however, each inspector tends to disagree more often about particular types of code smells.

Finally, we triangulate developer perceptions of code smells in IaC with their perceptions of smells in traditional source code. The results highlight the ambiguity of IaC code smell definitions and their co-occurrence relationships compared to smells in traditional source code. As such, 46.3% of IaC code smells are affected by inspector subjective interpretations, while only 1.6% of smells in traditional source code have annotation disagreements. This difference implies the necessity of enforcing strategies to align developer perceptions of IaC code smells before their attempt to annotate the oracle datasets.

In summary, our contributions are as follows.

- An empirical study that investigates the impact and prevalence of disagreements in the identification of IaC

code smells among developers. Furthermore, the study compares developer perceptions of code smells in traditional source code and IaC.
- An investigation of confounding factors, influencing the developer disagreements about code smells.
- A detailed replication package,[2] which includes the data and scripts to reproduce our results.

II. Background and Related Work

In this section we explain key concepts, discuss prior work, and summarize the original study of GLITCH [15].

A. IaC Management Technologies

IaC enables fast and continuous deployment through automating the provisioning of servers, configuration of hardware, and instant instantiating of servers through imaging [21]. Among others, three of the most popular and widely adopted IaC technologies are Ansible, Puppet, and Chef [19], [22]. All three technologies are open-source and offer enterprise and community-level support, with the possibility of adapting as cloud-based services [22]. Ansible and Chef code examples provide step-by-step imperative instructions to achieve desired end-states for software resources [23]; however, in Puppet, the end-states for resources are only declared, while the details for achieving those goals are decided by Puppet agent [22], [23].

B. IaC Security Code Smells

Similar to traditional source code, IaC can also be compromised by anti-patterns, decreasing their quality and causing them to be prone to defects and difficult to maintain overtime. If these anti-patterns affect IaC security-related design and implementation considerations, they are referred to as security code smells and make IaC code examples susceptible to the exploitation of their vulnerabilities [18]. Hence, it is important to detect security code smells in IaC, effectively and in a timely manner [4], [5], [24], [25].

C. Detection of Code Smells in IaC

Recent literature demonstrates a growing interest in identifying and removing code smells within IaC code examples [26], [27]. Applied research methods include qualitative analysis, pattern-matching, and machine learning methods.

Qualitative Analysis. Rahman et al. [4] studied the seven most frequent security code smell patterns in $1,726$ Puppet code examples. They also conducted similar studies in Ansible and Chef settings [5]. Rahman et al. [28] also studied Ansible scripts to detect categories of code smells that directly infect Ansible tasks. They conducted developer-focused surveys to identify improper practices that lead to task infection. Similarly, Bent et al. [23] surveyed developers to identify best practices for delivering high-quality Puppet code examples. Moreover, they created and validated a benchmark that ranks the quality of the code examples. Dalla Palma et al. [29] created a catalog of 46 metrics for evaluating the quality of Ansible code examples. These metrics are either obtained from general purpose programming languages or are specifically created for Ansible using the official documentation. Sharma et al. [20] analyzed $4,621$ Puppet repositories to identify incidences of code smells caused by lax design and configuration related practices. Bessghaier et al. [30] evaluated 82 Puppet examples for the prevalence of their co-occurring code smells, on both design and implementation levels. Furthermore, they investigated the impact of these code smells on the change and defect proneness of IaC. Drosos et al. [31], conducted a comprehensive analysis of 360 bugs in Ansible, Puppet, and Chef to understand their manifestation characteristics, underlying causes, reproduction requirements, and fixes.

Pattern-matching Methods. Rahman et al. [25] studied $1,448$ commit messages corresponding to Puppet code examples of 61 OpenStack repositories and used pattern-matching techniques to identify the eight most prevalent types of code smells in Puppet. Reis at al. [32] conducted a study with 131 practitioners and used their professional feedback to improve the performance of pattern-matching techniques previously designed for identifying code smells in Puppet code examples.

Machine Learning Methods. Other than using pattern-matching techniques, Rahman et al. [33] applied natural language processing techniques to the commit messages of OpenStack, Mozilla, and Wikimedia projects to identify incidences of code smells in IaC. Dalla Palma et al. [34] mined 104 GitHub repositories with defective Ansible code examples and used structural and evolution-based features to vectorize them. Embedding were then used to train machine learning classifiers for identifying different type of Ansible code smells.

D. Developer Perceptions of Code Smells

Prior work has investigated the developer perceptions of code smells present in traditional source code. For instance, Palomba et al. [11] studied whether Java developers perceive bad code designs as intentional decisions or consequences of undiscovered code smells. They also investigated developer perceptions of the severity of existing code smell symptoms. Chen et al. [12] studied Python code smells and compared them to code smells of compiled languages. Based on the discovered differences, Chen et al. introduced new metrics for detecting code smells in interpreted languages. De Bleser et al. [13] compared the code smells of Scala and JUnit testing frameworks. They surveyed 14 developers and evaluated whether they can correctly identify Scala code smells. The relationship between developer characteristics, including their education and professional role, and their perception of code smells was also investigated. Mello et al. [35] found that the expertise of developers shared a direct association with correct identification of code smell instances. On the other hand, Oliveira et al. [36] recommended to engage both novice and experienced developers for identifying smells. Sharma et al. [37] reviewed the literature to investigate the breadth of code smells including their characteristics, identification techniques, and prevailing inducing improper practices.

Inspired by aforementioned work, we are set to complement prior studies by investigating varied developer perceptions of

[2]The replication package is available at https://is.gd/6JcOqq.

TABLE I: The types of IaC security code smells identified by GLITCH (TLS: Transport Layer Security, Cryp. Alg.: Cryptographic Algorithms)

Security Code Smell Types	Description
Admin by Default	Default users get admin privileges.
Empty Password	Passwords are set as empty strings.
Hard-coded Secret	Secret information stored in text.
Unrestricted IP Address	IPs bound to '0.0.0.0'.
Suspicious Comment	Comments indicate defects/issues.
Use of HTTP without TLS	HTTP used without encryption.
No Integrity Check	No checksum for downloads.
Use of Weak Cryp. Alg.	Weak encryption algorithms used.
Missing Default in Case Statement	No default case in switch.

code smells in IaC—through analyzing discrepancies in the annotations of incidences of code smells.

E. Summary of the Original Study

Saavedra and Ferreira [15] introduced GLITCH to detect security code smells in Ansible, Puppet, and Chef.[3] GLITCH can detect nine categories of security code smells—the most common security code smell types across the studied technologies (see Table I). The tool uses an intermediate representation capable of abstracting similar IaC concepts in the supported technologies. For instance, the concept of a task in Ansible and a resource in both Chef and Puppet are mapped to the same atomic construct. The intermediate representation is derived from the original abstract syntax tree of the IaC code example under investigation. GLITCH's code smell detectors, which are based on regular expression rules, are defined on this intermediate representation, allowing the detection of code smells in a consistent fashion among IaC technologies [15].

Saavedra and Ferreira [15] also evaluated GLITCH's performance by creating and annotating an oracle dataset consisting of Ansible, Puppet, and Chef code examples. In total, seven graduate students with three years of experience in computer science, including software engineering, and one to two years of focused expertise in IaC and/or cyber security (i.e., a total of five years of expertise) annotated the oracle dataset of GLITCH. Saavedra annotated all of the code examples. The code examples of each IaC technology were annotated by two inspectors in addition to Saavedra, resulting in a total of seven unique inspectors. Closed coding [39] was applied to label the oracle dataset for the incidences of the curated and predefined catalog of IaC security code smells.

Three IaC developers inspected the code examples and annotated them independently—evaluating each line of code for existing code smell types. If the majority of the inspectors (i.e., at least two out of three) identified the same type of security code smell at the same line of code, the majority voting rule was enabled and the identified code smell instance and its location was included as an issue within the oracle dataset and the *agreement level* (i.e., 2/3 or 3/3) was reported. If the majority of the inspectors did not identify any code smell instances for a code example, that code example was labeled with *None* and was assigned with agreement level of 1/3. The original study showed that GLITCH outperforms prior approaches, achieving a mean improvement of 15.3 and 15.6 percentage points in precision and recall, respectively.

[3]GLITCH's most recent update also supports Terraform and Docker; however, their oracle dataset is not available [38].

III. Study Design

In this section, we discuss the insights derived from the annotation process of GLITCH and use them to outline our approach for conducting the study.

A. Developer Perceptions Insight

Based on the labeling process, three patterns emerge when GLITCH raises a *false alarm*—GLITCH identifies an incident of a certain type of security code smell while the majority of the inspectors disagree with what GLITCH has found:

- *Pattern 1 - None-smelly*: The strongest evidence that GLITCH is incorrect occurs when inspectors unanimously agree no smell exists at the target line of code.
- *Pattern 2 - Smelly with Majority Voting*: GLITCH can be considered incorrect when its detected code smell type differs from the smell type that is identified by the majority of the inspectors. In this scenario, a code smell is identified for a target line of code by at least two inspectors (i.e., agreement level $\geq$ 2/3); however, GLITCH identifies a different security code smell type for that line of code. Hence, although GLITCH is correct to raise an alarm at that location, the alarm type that it raises is likely to be incorrect.
- *Pattern 3 - Smelly with Uncertainty*: GLITCH can also be considered incorrect when the majority voting rule (i.e., agreement level $\geq$ 2/3) is not met. In such cases, there is a degree of uncertainty, as inspector perceptions of smells has not reached a consensus. As such, types of smells identified by inspectors are not the same—agreement level of 1/3. Nonetheless, as there is uncertainty among inspectors, GLITCH's false alarm may agree with the smell type identified by one of the inspectors.

Although all patterns of false alarms were treated similarly in prior work, we conjecture that the implications of these patterns are not the same. For example, while Pattern 1 - None-smelly shows strong evidence of a false alarm, Pattern 3 - Smelly with Uncertainty includes a degree of subjectiveness—caused by the disagreement of inspectors on their perceptions of code smells in IaC. Therefore, we set out to conduct an empirical study, using GLITCH's oracle dataset, to systematically re-evaluate the prevalence and characteristics of these patterns of false alarms for gaining a deeper understanding of the differences in developer perceptions of IaC code smells. Figure 1 provides an overview of our study design which is composed of *data acquisition* and *analysis*.

B. Data Acquisition

We use the replication package[4] provided by Saavedra and Ferreira [15] to access GLITCH's oracle dataset and annotations of individual inspectors of the oracle dataset. GLITCH's

[4]The replication package is available at https://is.gd/TnxhX5.

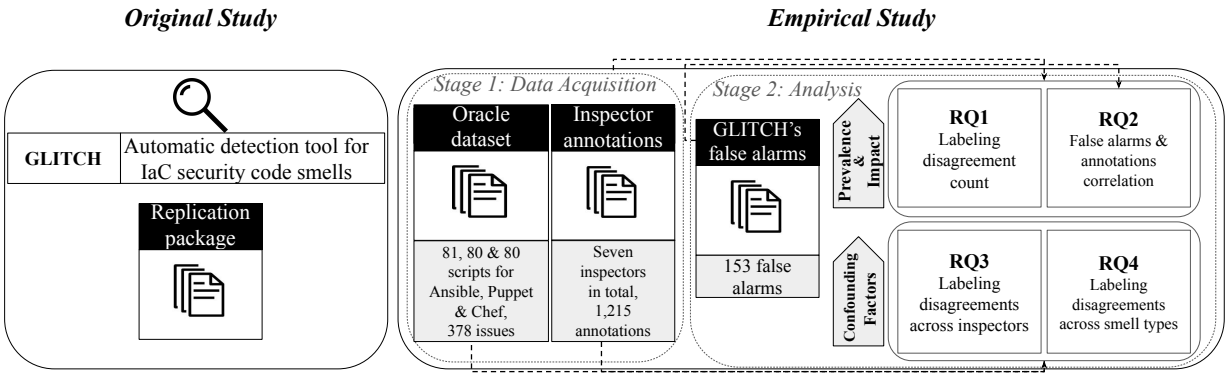


Fig. 1: Overview of the empirical study

TABLE II: Annotation count in GLITCH's oracle dataset

Ansible		Puppet		Chef	
Inspectors	*Count*	*Inspectors*	*Count*	*Inspectors*	*Count*
First Author	104	First Author	122	First Author	122
Insp. 1 (Ansible)	158	Insp. 1 (Puppet)	120	Insp. 1 (Chef)	190
Insp. 2 (Ansible)	114	Insp. 2 (Puppet)	106	Insp. 2 (Chef)	179
Total	376	Total	348	Total	491

oracle dataset consists of entries for Ansible, Puppet, and Chef and is created by annotating 81 code examples for Ansible and 80 code examples for each of Puppet and Chef settings for their incidences of code smells—241 code examples in total. We use the entire dataset to conduct our empirical analysis. Each code example is annotated by three different inspectors, resulting in a total of $1,215$ annotations—376 Ansible, 348 Puppet, and 491 Chef annotations, respectively (see Table II). As inspector annotations are carried out independently, their identified incidences of code smells may, or may not, be the same for a target line of code. If at least two inspectors annotate the same code smell instance for a target line of code (i.e., agreement level $\geq$ 2/3), that annotation is included as an issue within GLITCH's oracle dataset. From the total of $1,215$ annotations, 378 incidences of code smells are included as oracle dataset issues—113 Ansible, 117 Puppet, and 148 Chef issues, respectively. For each issue of the oracle dataset, we analyze the type of the smell incident, known as ground truth, and the reported number of inspectors who agreed on the annotation of the smell incident—known as agreement level. In contrast to the original paper of GLITCH [15], in which the agreement levels are reported as range-level averaging values, in the current paper we calculate the agreement levels using instance-level averaging. It is noteworthy that, we found errors in the original oracle dataset—discrepancies among few reported agreement levels and their corresponding inspector annotations. In cases of such discrepancies, we use the annotations made by the inspectors as the correct agreement level.

C. Analysis

We use the oracle issues and the individual annotations of the inspectors to analyze varied developer perceptions of IaC code smells, by assessing the prevalence of annotation disagreements and confounding factors which may influence them. Furthermore, we investigate whether these disagreements can undermine GLITCH's perceived performance.

Misalignment in developer perceptions: We analyze annotation disagreements among the inspectors of the oracle dataset as indications of misaligned developer perceptions of code smells in IaC. To achieve this, for each oracle issue we first determine the code example (*tainted script*) and its line of code (*tainted location*) that contains the code smell. For instance, Listing 1 is a tainted script in which an incident of the hard-coded secret smell appears on line 7 as the tainted location.

We then identify the annotators of the tainted script. If an inspector has not annotated any code smell for the tainted location, the individual annotation of that inspector for the current oracle issue is assigned as none. Otherwise, the types of code smells identified by the inspectors are assigned as their individual annotations of the target oracle issue. As such, for each oracle issue we can determine the inspector with varied smell perception compared to others.

GLITCH's false alarms: We analyze GLITCH's false alarms to investigate whether there is a relationship between developer perceptions of code smells and the tool's perceived performance—whether acclaimed false alarms are independent of the inspector annotations. To do so, we first identify the tainted script and the tainted location of the false alarm. We then determine whether there exists an oracle issue at the tainted location. If all inspectors unanimously agree the tainted location is not smelly, the false alarm belongs to Pattern 1 - Non-smelly. If by majority voting a code smell incident is annotated for the tainted location in the oracle dataset, the false alarm belongs to Pattern 2 - Smelly with Majority Voting. Otherwise, there is a degree of uncertainty among the inspectors when annotating the tainted location; hence, the false alarm belongs to Pattern 3 - Smelly with Uncertainty.

IV. Prevalence & Impact of Disagreements

In this section, we quantify the prevalence of disagreements among inspector perceptions on IaC code smells.

RQ1: How prevalent are labeling disagreements?

Approach. To address this research question, we analyze IaC code smells that are agreed upon by the majority of the

TABLE III: Uncertainty count in identifying IaC code smells

	Ansible	Puppet	Chef
Agreement Level of 1/3	4	6	2
Agreement Level of 2/3	45	51	67
Agreement Level of 3/3	64	60	79
Total	113	117	148

inspectors. For each issue in the oracle dataset, we check the annotations of individual inspectors. If casted labels of individual inspectors are the same, we interpret the oracle issue as an agreement. Cases where all three inspectors are in agreement are labeled as agreement level of 3/3. Cases where two inspectors are in agreement are labeled as agreement level of 2/3 while cases without any agreement among inspectors are labeled as agreement level of 1/3.

Observation 1: **Approximately half of the studied oracle issues have some degree of disagreement among the inspectors.** Table III shows that 64 of 113 (56.6%), 60 of 117 (51.3%), and 79 of 148 (53.4%) of the studied oracle issues are at agreement level of 3/3 (unanimous agreement across inspectors) in Ansible, Puppet, and Chef, respectively. In other words, 43.4%–48.7% of oracle issues have at least one dissenting vote. Presence of annotation disagreements for nearly half of the oracle issues demonstrates that there may be ambiguity in developer perceptions of code smells in IaC. To investigate whether there are any meaningful differences between dissenting votes at different agreement levels, we apply *Cochran's Q* statistical test [40] for each IaC technology. This test and any other conducted statistical tests are non-parametric as we do not make any assumptions regarding the probability distribution of the data under investigation. When running multiple comparisons, to raise the confidence bar, we apply the Bonferroni correction [41] on the threshold of significance—α/k, with α being the confidence limit = 0.05 and k being the count of testing hypotheses which is three when running the test across Ansible, Puppet, and Chef settings. For Cochran's Q test we correct the significance threshold as $0.05/3 = 0.017$. We use the values in Table III, as input effect sizes to evaluate whether annotation disagreements among inspectors are statistically significant or the disagreements among inspectors are caused by chance. Based on the test result (i.e., $p - value > 0.017$), we conclude that there are no statistically significant differences among the inspector annotation disagreements at different agreement levels.

Observation 2: **3% of the studied oracle issues are without agreement among inspectors.** Table III reveals that 109 of 113 (96.5%), 111 of 117 (94.9%), and 146 of 148 (98.6%) of the studied oracle issues of Ansible, Puppet, and Chef, have an agreement level of at least 2/3 among the inspectors; however, there are 12 cases (3.2%) without any agreement among inspectors—they have agreement level of 1/3 (see section III). The presence of cases without any agreement highlights the possibility of existing fully varied developer perceptions on code smells in IaC. As such, relying solely on the majority voting for determining definitive ground truth may lead to potential flaws in the oracle datasets; however, oracle issues with one dissenting vote represent a group of fully-biased incidences of code smells in IaC.

> ***RQ1*** **Summary.** Nearly half (i.e., 43.4%–48.7%) of the studied IaC oracle issues have some degree of disagreement among inspectors, suggesting that there is subjectivity at play, which should be handled with care during subsequent analyses.

RQ2: Do false alarms correlate with the annotations of the oracle dataset?

Approach. To address this research question, we analyze the occurrences of GLITCH's false alarms. For these cases, the tool always receives the blame; however, in cases of disagreements among the inspectors, it may be possible that GLITCH's raised alarm is similar to one of the inspector's identified smell type (see section III). To further analyze this possibility, for each false alarm we examine corresponding annotations of individual inspectors to determine whether any incident of smell is identified by the inspectors and if true, what is the identified code smell type. Through this analysis, we aim to measure the strength of correlation (or lack thereof) between GLITCH's false alarms and the types of IaC code smells that are identified by the individual inspectors or are agreed upon among the majority of them. This evaluation enables us to assess the impact of varied and ambiguous developer perceptions of code smells in IaC on the perceived performance of smell detection tools like GLITCH.

Observation 3: **Roughly half of the false alarms are raised in code for which inspectors raised concerns.** In Ansible, Puppet, and Chef settings, 6 of 15 (40%) , 54 of 86 (62.8%), and 22 of 52 (42.3%) GLITCH's false alarms belong to Pattern 1 - None-smelly, respectively. This suggests that for 71 of 153 (46.4%) of false alarms, GLITCH is not entirely incorrect—it detects an IaC code smell with at least one inspector agreeing on the presence of an IaC code smell instance.

Observation 4: **Approximately a quarter of false alarms are misidentified for other types of code smells in IaC.** In Ansible and Puppet settings, 5 out of 15 (33.3%) and 18 of 86 (20.9%) false alarms are identified as smells by the majority of inspectors; however, the smell type differs from that of the tool—Pattern 2 - Smelly with Majority Voting. Similarly in Chef setting, 16 of 52 (30.8%) false alarms also belong to Pattern 2 - Smelly with Majority Voting. This suggests that while GLITCH correctly detects an incident of a security code smell, it is prone to mistakenly applying the wrong type. This is not unexpected as GLITCH uses regular expression detection patterns for identifying code smell types. Due to the nature of different smell types and their close similarities, their mapped detection patterns may include mutual rules. For instance, this is the case for hard-coded secret and admin by default code smell types. As such, if the tool detects a smell incident using a regular expression rule that is mutually used among hard-coded secrets and admins by default, it may associate the incident with the wrong smell pattern—the present code smell type is admin by default; but, the tool identifies it as hard-coded secret. The implications for such false alarms

Listing 2: Part of a real-world Redhat Puppet code example

```
1 class galera::server(
2 $bootstrap = false,
3 $debug = false,
4 $wsrep_node_address = undef,
5 $wsrep_provider='usr/lib64/galera/libgalera_smm.so',
6 $wsrep_cluster_name    = 'galera_cluster',
7 $wsrep_cluster_members = [ $::ipaddress ],
8 $wsrep_sst_method      = 'rsync',
9 $wsrep_sst_username = 'root', )
```

are different than Pattern 1 - None-smelly cases, where all inspectors agree that there is no smell; but, the tool reports one. Pattern 2 - Smelly with Majority Voting false alarms discriminate between smell-prone and clean code, which may be a boon for project maintainers who can act on such coarse grained information when, e.g., planning refactoring efforts; however, the misleading reports of incorrect code smell types can still be harmful and erode the trust of users in the tool.

Moreover, we conduct an open coding analysis to investigate any potential relationship among IaC block context and code smell incidences. Specifically, the first and the third author of the paper evaluate each false alarm raised by GLITCH, to (1) discover whether the false alarm is raised inside a block (i.e., a number of Ansible tasks or resources in Puppet and Chef, grouped together to accomplish a certain goal) and (2) to identify the goal of the surrounding IaC context. Based on the results of the analysis, 26.7% (4 out of 15), 10.5% (9 out of 86), and 38.5% (20 out of 52) of the false alarms in Ansible, Puppet, and Chef, reside outside of IaC blocks—for all investigated technologies, the majority of false alarms are raised within IaC blocks. Similarly, 33.30% (5 out of 15), 32.6% (28 out of 86), and 50.0% (26 out of 52) of false alarms in Ansible, Puppet, and Chef, are raised within IaC enforcing different levels of user privileges. 80.0% (4 out of 5), 100.0% (28 out of 28), and 76.9% (20 out of 26) of these false alarms are raised inside Ansible, Puppet, and Chef IaC blocks, respectively. These results suggest that identifying the correct type of IaC code smells can be particularly challenging when IaC is used to set different levels of privileges for users.

Observation 5: **More than four-fifths of misidentified hard-coded secrets by GLITCH are admins by default.** In Ansible setting, five false positives are misclassified as hard-coded secrets by GLITCH; however, the ground truth of four of them is admin by default while the last false positive has an empty password as its ground truth. In Puppet setting, 18 false positives are misclassified as hard-coded secrets by GLITCH, despite the ground truth indicating five cases as empty password and 13 cases as admin by default. A similar pattern occurs in Chef, where 15 false positives are misclassified as hard-coded secrets by GLITCH, although their ground truth labels point to admin by default. In total, 32 of 38 (84.2%) hard-coded secret false alarms of GLITCH are labeled as admin by default in the oracle dataset. For instance, Listing 2 illustrates such false alarms in the Puppet code of *Redhat OpenStack*.[5] In this example, line 9 sets the username of the server as root which is an indicator of admin by default—similar to the ground truth label. Meanwhile, GLITCH recognizes the code smell incident as a hard-coded secret. Current observations suggest that the definitions of different types of code smells in IaC may not be interdependent which will lead to ambiguities with IaC practitioners and pose challenges for relevant detection tools such as Checkov [42] and Terrascan [43]. Instances of one type, e.g., type A, can be regularly misidentified as another type—type B. This suggests the need for expert attention to determine if type A and type B can co-occur.

[5]The code example is accessible at https://is.gd/bnofu4.

> ***RQ2*** **Summary.** Roughly half (46.4%) of the false alarms raised by GLITCH are not entirely incorrect, with at least one inspector agreeing on an incident of a smell, suggesting difficulties in current developer perceptions of code smells in IaC.

V. Investigating Confounding Factors

In this section, we investigate to what extent common confounding factors can explain the incidences of disagreement among developer perceptions on code smells in IaC.

RQ3: How labeling disagreements distribute among inspectors?

Approach. In this research question, we aim to understand how labeling disagreements are distributed among the inspectors who took part in the labeling process. As such we can determine whether an individual inspector has a different perception on a specific code smell type from the rest. To achieve this goal, we compute how often each inspector submits a dissenting vote at each agreement level for each IaC technology. More specifically, we first select oracle issues with an agreement level other than 3/3. Then, for each selected instance, we identify which inspectors (i.e., first author, Inspector 1, or Inspector 2) cast the dissenting vote. Using these issues, we analyze the frequency of dissenting votes at agreement levels of 1/3 and 2/3 to study whether inspector tendencies are at play. In cases of agreement level of 1/3, the dissenters are the sole vote in favour of the code smell instance being present. At agreement level of 2/3, dissenters vote against the majority, either in terms of the presence or the type of IaC code smell. In both cases, the dissenting vote is overruled by the majority of the inspectors.

Observation 6. **All inspectors cast dissenting votes about code smells in IaC.** Table IV shows how often each inspector is implicated in IaC code smell perception disagreements. For Ansible, the first author, Inspector 1, and Inspector 2 are implicated in 24.5%, 36.7%, and 38.8% of disagreements. For Puppet, the first author is implicated in only 15.8% of disagreements while Inspector 1 and Inspector 2 dominate the disagreements by casting 36.8% and 47.4% of the dissenting votes, respectively. For Chef, the first author casts 59.4% of the annotation disagreement votes while Inspector 1 and Inspector 2 are implicated in only 23.2% and 17.4% of disagreements, respectively. At agreement level of 2/3, it is clear that all inspectors can be the one who cast the dissenting vote. Indeed,

TABLE IV: The frequency of dissenting votes casted by inspectors across different agreement levels

Agreement Level	Ansible			Puppet			Chef		
	First Author	*Insp. 1*	*Insp. 2*	*First Author*	*Insp. 1*	*Insp. 2*	*First Author*	*Insp. 1*	*Insp. 2*
1/3	3	0	1	5	0	1	2	0	0
2/3	9	18	18	4	21	26	39	16	12

while Inspectors 1 and 2 cast more dissenting votes in Ansible and Puppet settings, the situation is reversed in Chef setting, where the first author of the original study contributes to 39 dissenting votes, whereas Inspectors 1 and 2 cast only 16 and 12, respectively. Since all inspectors are implicated in disagreements, the dissenting votes cannot be simply explained by a (single) rogue inspector who always has a different perception of code smells in IaC from the rest.

To further analyze the inspector implications in annotation disagreements, we apply *Fisher's Exact* statistical test [44] and evaluate probable significant differences between dissenting votes casted by the inspectors. For each IaC technology, the test's contingency table is constructed among every two inspectors, using the count of their dissenting votes and oracle issues they agree on. For this test we correct the significance threshold to 0.05/3 = 0.017. Based on the results, there are no significant annotation differences (i.e., $p - value > 0.017$) among the inspectors of Ansible. Meanwhile, there exist significant differences among the first author and Inspector 2 in Puppet and the first author and Inspectors 1 and Inspector 2 in Chef (i.e., $p - value < 0.017$), respectively.

We also analyze the distribution of annotation disagreements among the inspectors by calculating the diversity measure similar to the study conducted by Moussa et al. [45]. For this purpose, for every IaC technology we investigate the disagreement diversity among every inspector pair—three values for three inspector pairs; the first author with Inspector 1 and 2, and Inspector 1 with Inspector 2. The diversity measure is calculated as the fraction of oracle issues the pair annotates differently, compared to the total population of the oracle issues for each IaC technology. As such, the diversity measure varies from 0 to 1 with higher values indicating higher levels of annotation disagreement among the inspector pairs. In this context the diversity among the three inspector pairs are 0.32, 0.33, and 0.37, 0.33, 0.38, and 0.44, and 0.42, 0.40, and 0.26 for Ansible, Puppet, and Chef settings, respectively. According to the results, Ansible inspectors have the most similarity when annotating the oracle dataset (i.e., mean diversity measure = 0.34) while inspectors of Puppet demonstrate the highest annotation diversity—mean diversity measure = 0.38. Nonetheless, these results demonstrate similar levels of inspector diversity among IaC technologies.

Observation 7. **The first author is implicated in almost all of the disagreements at the agreement level of 1/3.** Among the incidences of code smells with agreement level of 1/3, 12 of them are included as issues within the oracle dataset. In these cases, one inspector cast the dissenting vote in favour of the occurrence of the oracle issues. Table IV shows the break down of such dissenting votes per inspector—the row for agreement level of 1/3. These occurrences are not distributed equally among all inspectors. 10 of 12 (83.3%) of such oracle issues have the sole favoring dissenting vote from the first author—the first author's annotation is the ground truth. There are only two cases, one for Ansible and one for Puppet, where Inspector 2 submits the ground truth annotation, indicating different developers cast different votes towards the occurrence of varied code smell types. Similar observation is made for GLITCH's false alarms that belong to Pattern 3 - Smelly with Uncertainty. A false alarm can be raised when only a single inspector is in favor of the smell incident, indicating the tool's false alarm could be considered a code smell, if the oracle issue was inspected by different developers.

> ***RQ3*** **Summary.** Although the first author casts the majority of the dissenting votes at agreement level of 1/3, the results of the statistical tests demonstrate that all inspectors cast a considerable number of dissenting votes towards the incidences of IaC code smells.

RQ4: How are labeling disagreements distributed across the types of IaC code smells?

Approach. In this research question, we aim to understand how labeling disagreements are distributed across different types of IaC code smells. Specifically, we want to know whether there is a specific IaC code smell type for which developers have different perception from each other. To achieve this goal, for each IaC technology, we compute how often each inspector submits a dissenting vote against the majority and analyze the IaC code smell types that receive the dissenting votes. Specifically, we first select oracle issues with agreement level of 2/3. Then, for the selected issues, we determine the inspector who is responsible for submitting the dissenting vote. As such, we are able to calculate the contributions of individuals in the submission of dissenting votes stratified by code smell types and IaC technology settings. Additionally, we aim to determine whether varied code smell types contribute to similar uncertainties.

Observation 8. **No single type of IaC code smells dominates the disagreements.** Figure 2 illustrates the misidentification of different types of IaC code smells among the inspectors at agreement level of 2/3. There is no single type that is responsible for the bulk of dissenting votes. Specifically, in Ansible and Puppet settings, the two most frequently occurring types (i.e., use of HTTP without TLS and admin by default) only account for 10 of 45 (22.2%) and 12 of 51 (23.5%) of the incidences of dissenting votes, respectively. On the other hand, in Chef setting, admins by default account for 27 of 67 (40.3 %) of the incidences of dissenting votes—suggesting that this code smell type is particularly vague in the context of

Chef, leading to varied developer perceptions. One reason we found is that the first author missed the incidences of the smell when mysql commands are used with the root user, which have frequent appearances in Chef compared to other IaC settings. Nonetheless, the remaining 40 of 67 (59.7%) disagreement incidences are rather evenly dispersed among the other six code smell types, indicating that no single type of IaC code smells dominates the annotation disagreements.

Additionally, we calculate Shannon's normalized entropy [46] to assess the level of uncertainty within IaC settings. The entropy values are calculated based on the number of votes for each smell type for each IaC setting (e.g., for Ansible, the voting entropy for each smell type is [6/28, 5/28, 1/28, 5/28, 1/28, 10/28], with a total of nine smell types), spanning the range between 0 and 1—with higher entropy values indicating that the dissenting votes are not localized to a specific code smell type. In this context, the entropy values are 0.7, 0.8, and 0.6 for Ansible, Puppet, and Chef settings, respectively. It is noteworthy that the entropy is high across all settings; however, compared to others, Chef has the lowest entropy, indicating that the majority of its dissenting votes span across fewer code smell types. Nonetheless, these entropy values confirms the dispersion of inspector annotation disagreements across different IaC code smell types.

Finally, we conduct *Eta-Squared* statistical test [47] as a measure of effect sizes to analyze the extent of which the count of dissenting votes vary among different code smell types. For this purpose, for each IaC setting, we calculate the correlation ratio between variances of dissenting votes for each code smell type and the total variance of disagreements across all smell categories. As such the correlation ratio ranges from 0 to 1, with values equal to or greater than 0.14 as indicators of large effect sizes of code smell types on the count of their corresponding dissenting votes [48]. Similarly, correlation values less than 0.14 but greater than or equal to 0.06 indicate medium effect sizes while values less than 0.06 are indicators of small effect sizes. According to the results of the conducted analysis, the use of HTTP without TLS (i.e., $\eta^2 = 0.11$) in Ansible and admins by default (i.e., $\eta^2 = 0.09$) in Puppet have medium effect sizes on their corresponding dissenting votes. Furthermore, the only large effect size (i.e., $\eta^2 = 0.23$) exists for admins by default in Chef. As shown in Figure 2, the same code smell type can have varying levels of inspector discrepancy across IaC settings—small, medium, and large effect sizes for admins by default across Ansible, Puppet, and Chef, respectively. To identify underlying reasons for these differences across technologies, we compare disagreement cases of the smell across them. Our analysis demonstrates that the Ansible user defined to execute a play is a common cause of discrepancy in admins by default detection. Meanwhile, Shell operations in Puppet raise disagreements about the smell. Finally, in Chef, SQL commands are causing disagreements. As such, we can conclude that inherent differences among IaC settings enforce the developers to adopt varying best practices accordingly, causing different levels of understanding and disagreement rates towards the same code smell type across settings.

Observation 9. **Different developers have varied tendencies for misidentifying different types of IaC code smells.** Figure 2 shows the number of misidentifications for individual inspectors across different types of code smells in IaC. The figure also highlights that different inspectors have varied biases toward certain IaC code smell types. For example, the first author of the original study submits 9 of 10 and 11 of 11 of the dissenting votes for use of HTTP without TLS code smell instances in Ansible and Chef settings, respectively. A similar pattern is also found in Puppet setting, with Inspector 2 submitting 10 of 10 of the dissenting votes for missing default in case statement code smell instances. This figure also highlights that some inspectors do not submit any dissenting vote for certain types of IaC code smells. For instance, the first author never submits a dissenting vote for missing default in case statement across all settings. This suggests that deep subject matter expertise may help to reduce rates of dissension or varied perceptions of a specific type of code smell in IaC.

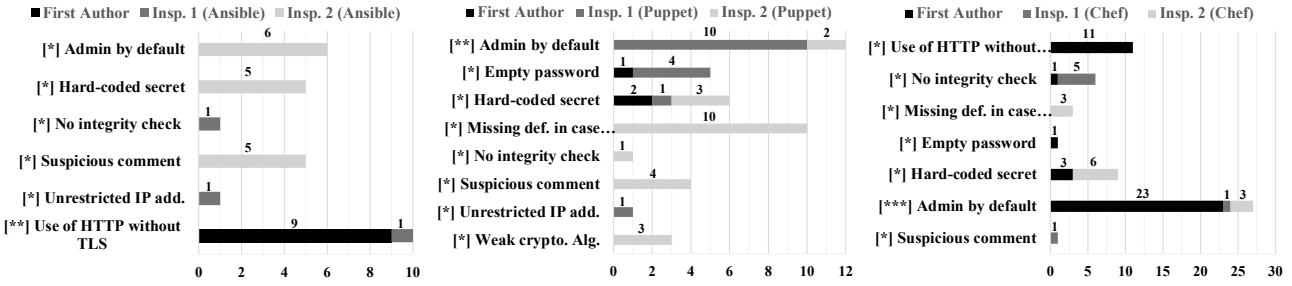
> ***RQ4*** **Summary.** There is no single type of IaC code smell that is responsible for the bulk of the disagreements; however, different inspectors exhibit varied tendencies in submitting dissenting votes when the types of smells are taken into consideration.

VI. Practical Implications

In this section, we discuss the practical implications for developers and researchers, derived from the results in section IV and section V.

PI1: Manual identification of code smells in IaC is profoundly impacted by inspector bias.

Several observations from our research questions show the profound impact of inspector bias on the oracle dataset.
(1) Half of the IaC oracle issues depend on the chosen inspectors (RQ1 - Observation 1). Nearly half of the studied IaC oracle issues have some degree of disagreement among inspectors. As such, in case of an oracle issue with agreement level of 2/3, if one of the two majority inspectors is replaced with a new developer, whose perception of IaC code smells is similar to the dissenter, the target issue would be removed from the oracle dataset, subjecting the oracle dataset to change.
(2) Subjectivity towards different types of IaC code smells varies among developers (RQ3 - Observation 6 and RQ4 - Observation 9). All inspectors regularly cast dissenting votes about IaC code smells; however, one (e.g., inspector X) may constantly misidentify a specific type of code smell (e.g., type A) for another category—type B. As such, if inspectors share their knowledge before annotation, inspector X can be influenced by others and their perceptions of code smells may align. This could consequently produce a more robust oracle dataset with increased agreement levels.
(3) The extent to which developers neglect incidences of code smells varies among them (RQ1 - Observation 2). 12 issues of the IaC oracle dataset have agreement level of 1/3. Six of these issues are cases where two inspectors do not agree on the code smell type, while the last inspector does not identify



(a) Ansible (b) Puppet (c) Chef

Fig. 2: Inspector implications in code smell misidentification for each code smell type, except none

Eta-Squared statistical test, [*]: small effect sizes ($\eta^2 < 0.06$), [**]: medium effect sizes ($0.06 <= \eta^2 < 0.14$), and [***]: large effect sizes ($\eta^2 >= 0.14$))

Listing 3: Part of a real-world OpenStack Puppet code example

```
1 class { '::designate::api':
2     listen           => "${::openstack_integration::
          config::ip_for_url}:9001",
3     api_base_uri     => "http://${::openstack
      _integration::config::ip_for_url}:9001",
4     auth_strategy    => 'keystone',
5     enable_api_v2    => true,
6     enable_api_admin => true}
```

any incidences of code smells. The remaining six issues with agreement level of 1/3 (six out of 12) are only identified by the first author while other two inspectors do not identify any incidences of code smells. Listing 3 demonstrates one instance of these six issues in a Puppet code example for OpenStack integration.[6] In this example, the URL address is accessed without establishing any security layer (line 3) and thus the occurrence of use of HTTP without TLS is clear. Meanwhile, Inspector 1 and Inspector 2 overlooked the smell incident, despite the problem being trivial to identify. Incidences of code smells in the rest of these six issues are also easily detectable, if inspector 1 and inspector 2 were more diligent annotators, consequently establishing an oracle dataset with increased agreement levels.

For Developers. Based on discussed observations, we argue that the use of the majority voting rule can reduce the impact of developer personal biases when annotating an oracle dataset; but, it does not eliminate it completely. To address this problem, first, based on insights derived from previous studies [49], [50], we make recommendations for the participation of more than three developers in the annotation of the IaC oracle dataset. As such, the impact of subjectivity towards different IaC code smell types among different inspectors decreases. Second, we make recommendations for the inspectors to share their knowledge before annotating the oracle dataset. This recommendation aims to align their varied perceptions of IaC code smells. Finally, we make recommendations for a meeting after the annotation, to identify potential cases of negligence and reach consensus in such disagreements. As demonstrated by Oliveira et al. [36], collaboration improves the effectiveness of code smell identification in traditional source code. Thus, we expect the collaborative approach to also improve the effectiveness of annotating the oracle dataset of smells in IaC.

[6]The code example is accessible at https://is.gd/EXloIs.

PI2: IaC code smell types can co-occur.

Several observations from studied research questions support co-occurrences of varied types of IaC code smells.

(1) Employed rules by GLITCH for identifying admins by default and empty passwords overlap with identification rules for hard-coded secrets (RQ2 - Observation 4 and RQ2 - Observation 5). With the exception of a single false alarm in Chef setting (15 out of 16), all incidences in which GLITCH misidentifies one type of IaC code smell as another involve the misidentification of admin by default oracle issues as hard-coded secret false alarms. The same pattern is observed for Ansible and Puppet settings, with GLITCH regularly misidentifying admins by default or empty passwords as hard-coded secrets. By further analyzing GLITCH's output, we find that in all of these cases GLITCH generates two alarms for the same location: not only it generates the false alarm (i.e., hard-coded secret), but also it generates the ground truth—either admin by default or empty password.

(2) Inspectors have ambiguous perceptions of admins by default, empty passwords, and hard-coded secrets (RQ4 - Observation 9 and Figure 2). Inspectors also demonstrate difficulties in distinguishing hard-coded secrets from admins by default and empty passwords when annotating the oracle dataset. Our analysis reveals eight annotations where inspectors misidentify an admin by default or an empty password as a hard-coded secret. For instance, in Listing 2, the first author and Inspector 2 assign an admin by default smell to the root username on line 9 while Inspector 1 misidentifies it as a hard-coded secret. For half of these annotations (four out of eight), the inspectors also identify the correct code smell type.

An underlying reason for difficulties in distinguishing these smell types from each other is related to their nature. If in a code example, the user is declared as root (i.e., $user = "root", both hard-coded secret and admin by default smell incidences exist in the code example. Similarly, if the password is declared as an empty string (i.e., $password = ""), both hard-coded secret and empty password smell incidences exist. Meanwhile, not all incidences of admin by default, or empty password, are perceived as hard-coded secrets. If root usernames, or empty passwords, are declared using dynamic

TABLE V: The types of Java code smells identified in *MLCQ* dataset

Java Code Smell Types	Description
Blob	A class with too many diverse functionalities.
Data Class	A class with only data attributes.
Feature Envy	A method using other classes' members.
Long methods	A method with too many responsibilities.

variables instead of literal strings (e.g., $passOne = “”, *$password = $passOne*) they would be only perceived as admins by default, or empty passwords, and not hard-coded secrets.

For Researchers. Considering these findings, we make recommendations for researchers to revise the current catalog of IaC code smells. The new catalog should set guidelines on the co-occurrence of incidences of admin by default and empty password with hard-coded secrets. If incidences of admin by default, or empty password, and hard-coded secret occur simultaneously, only the former should be perceived as the ground truth. As such, the count of generated false alarms would decrease and inspector uncertainties would reduce. For instance, GLITCH raises 11, 57, and 37 hard-coded secret false alarms for Ansible, Puppet, and Chef settings, respectively. By applying our previous recommendation, co-existing hard-coded secret false alarms with admin by default, or empty passwords, would not be triggered and the count of false alarms would be reduced by a total of 67 instances, increasing GLITCH's precision from 0.42 to 0.57, 0.14 to 0.20, and 0.20 to 0.29 in Ansible, Puppet, and Chef settings, respectively.

VII. Triangulation

In this section of the paper, we triangulate developer perceptions of code smells in IaC with code smells in traditional sourced code. Specifically, we compare the prevalence of code smells between IaC, as the *target domain*, and traditional source code as the *out-of-context (OOC)* domain. We choose Java code smells as the OOC domain as (1) Java is one of the most popular programming languages for writing traditional source code [51], [52] and (2) code smells in Java are previously investigated [11]. We use *MLCQ* dataset provided by Madeyeski and Lewowski [53] to access Java code smells and individual annotations of the inspectors. MLCQ dataset was created by annotating 792 Java projects for four types of Java code smells including *blob* (or *God class*), *data class*, *feature envy,* and *long methods* (see Table V). For this purpose, 26 developers used closed coding and annotated $4,770$ unique Java code examples, resulting in total of $14,729$ annotations.

Approach. Similar to the conducted analysis for GLITCH's IaC oracle dataset, We analyze MLCQ dataset for developer perceptions of Java code smells and possible differences among their understanding of varied Java code smell types. We specifically, investigate the prevalence of annotation disagreements among MLCQ inspectors. For this aim, we first identify unique Java code examples with at least two different inspectors—$1,235$ out of $4,770$. From these unique code examples, we then identify the ones which are assigned with different code smell types by the inspectors. These different assigned labels are annotation disagreements among MLCQ inspectors and are indicators of misalignment in developer perceptions of Java code smells. Finally, we triangulate the prevalence of annotation disagreements among MLCQ inspectors with our results on IaC to infer whether there are any significant differences between developer perceptions of code smells in traditional source code and IaC.

Observation 10. **98.4% of incidences of Java code smells are agreed upon by all of the participating inspectors.** Out of $1,235$ code examples with at least two different inspectors, 20 of them are assigned with different annotation labels. In other words, in only 1.6% of the cases, developer perceptions of Java code smells are different from each other.

Observation 11. **When misidentified, incidences of blob are only misinterpreted for data class incidences.** Half of the detected annotation disagreements (10 out of 20) are between incidences of blob and data class Java code smell types, while the rest (10 out of 20) are among the incidences of feature envy and long methods. The consistency in misidentifying Java code smell types suggests that there may be similarities among blob and data class (also among feature envy and long methods) that render them challenging for developers to differentiate.

Implication 1: There is a significant difference between developer perceptions of code smells in IaC and traditional source code.

Observation 10 demonstrates the prevalence of Java code smell disagreements to account for 1.6% of the investigated unique code examples. Meanwhile, 46.3% of studied IaC incidences have at least one dissenting vote. This contrast in the prevalence of disagreements could be an indicator of varied developer perceptions of code smells in IaC and traditional source code. For investigating the validity of this assumption, we use Fisher's Exact statistical test [44]. The test is conducted to evaluate whether there exists statistically significant association between two groups of data with small distributions—code smell incidences with and without disagreements. As such, the total count of code smell incidences in Java and IaC constitute the rows of the contingency table—$1,235$ and 378 incidences for Java and IaC settings respectively. The columns of the contingency table are the counts of dissenting votes (i.e., 20 and 175 votes for Java and IaC, respectively) and the counts of agreements towards the presence of code smells. The test result demonstrates a significant difference (i.e., $p - value < 0.05$) among the prevalence of developer disagreements and their perceptions of code smells in IaC and traditional source code.

Implication 2: Compared to IaC, code smells in traditional source code are less ambiguous.

Observation 11 states that each Java code smell type is only misidentified as one other specific type—blob incidences are only misinterpreted for data class and incidences of feature envy are only misidentified as long methods. Meanwhile, that is not the case for IaC code smells. Other than two specific code smell types that are regularly misidentified as each other (i.e., empty passwords, or admins by default, and hard-coded secrets), IaC code smell types are also misidentified as none,

indicating the identification of the correct code smell type by only one inspector (see section VI). Based on these observations, we argue that code smells in traditional source code are less ambiguous compared to code smells in IaC. Hence, it is much easier for developers to correctly differentiate among Java code smell types.

One of the reasons for this difference can be due to the definitions of Java code smells. Studied Java code smells are defined based on object-oriented programming paradigm. Each definition clearly declares the hierarchy level that the incident of the smell can occur at—whether the code smell affects a class or a method of a class (see Table V). As such, it is unlikely for developers to misidentify the incidences of code smells across different hierarchy levels. On the other hand, the definitions of IaC code smells are not hierarchical in nature. For instance, Ansible resources can be grouped into tasks (i.e., method-level) and blocks—class-level. Meanwhile, IaC code smell definitions, with the exception of missing default in case statement which is inherently method-level, do not determine the resource level that the code smell incident can occur at. This ambiguity makes it challenging for developers to accurately differentiate the types of code smells in IaC, in contrast to code smells in Java.

It is also possible that the higher agreement among Java developers is due to their higher level of expertise compared to GLITCH's inspectors. To evaluate this assumption we need to analyze whether there exists any meaningful relationship among the inspector levels of expertise and the count of their casted dissenting votes. According to the original study [15], the inspectors of the IaC oracle dataset are Master students with five years of experience—constituting three years of experience in computer science, including software engineering, and one to two years of focused expertise in IaC and/or cyber security fields. Consequently, it is not possible to conclude whether IaC inspector levels of expertise affect their tendencies for casting dissenting votes. Meanwhile, the levels of expertise of Java code smell inspectors varies among them. We use the MLCQ inspector background survey to investigate, using *Chi-Square* statistical test [54], whether there exists any correlation between Java inspector levels of expertise and their tendencies to cast dissenting votes. The results of the test demonstrates no significant correlation (i.e., $p - value > 0.05$) between the levels of expertise of the inspectors and the likelihood of casting dissenting votes towards Java code smells.

VIII. Threats to Validity

We breakdown the threats to the validity of our study into construct, internal, and external.

A. Threats to Construct Validity

These threats pertain to the extend to which our measurements are accurately captured. We use a simple fraction to demonstrate the labeling agreement among the inspectors of the IaC oracle dataset. For instance, if two out of three inspectors identify the same incident of IaC code smells at a line of code, the level of labeling agreement among them is 2/3. Meanwhile, there are other practices (e.g., Cohen's Kappa and Fleiss' Kappa coefficients) to measure the agreement among two or multiple inspectors, respectively [55]–[57]. We do not use these metrics because the annotation process of the IaC oracle datasets is not aligned with their calculation. Kappa coefficients are calculated using the total number of inspected observations. In our case, the developers inspect every line of code examples. If we strictly follow the definition, the Kappa coefficients are calculated using the number of code lines and thus the metrics would be strongly biased towards full agreement among the inspectors because most of the lines in the oracle dataset are easily identified as not smelly. Consequently, neither of these coefficients would reflect disagreements among the developers when they identify the presence of code smells, but disagree about their type. Hence, we report proportions of disagreement to measure the consensus among developer perceptions when they identify incidences of code smells.

B. Threats to Internal Validity

These threats are associated with inherited challenges threatening the validity of our methods. In the current study, we use the previously annotated oracle dataset of GLITCH. Except for Saavedra, none of the other authors were involved in the annotation process of the oracle dataset of GLITCH. Furthermore, at the time, the annotation of the oracle dataset was only carried out to serve as an evaluation baseline for developing GLITCH as a detection tool for IaC code smells. Hence, our analysis is conducted on the perception of IaC developers who are external to the current study. As such, the results of our study are ensured not to be affected by examiner bias. For conducting the triangulation, we use MLCQ and GLITCH's oracle datasets annotated by different developers which may have caused the observed differences among developer perceptions of code smells in traditional source code and IaC. Although we acknowledge this possibility, it is highly unlikely to find developers who are experts in both traditional source code and IaC [58]. Consequently, we have opted to use oracle datasets that are annotated by developers who are experts in their own respective fields. Furthermore, by recruiting experts for annotation, although the inspectors of oracle datasets are different from one another, they are similar with respect to their level of specific expertise. Finally, one may argue that the population of GLITCH's oracle dataset is not adequate for conducting our empirical study. We refute this concern by emphasizing that the developer perceptions of IaC code smells is analyzed using their individual annotations of code examples—$1,215$ annotations.

C. Threats to External Validity

These threats relate to generalizability of our results. The analysis of IaC code smells is based on the oracle dataset of GLITCH which is specifically designed for Ansible, Puppet, and Chef settings. Hence, our results may not be applicable to other IaC technologies including Terraform [59], which is a widely used and popular IaC technology. Furthermore, the oracle dataset of GLITCH does not represent smells in IaC scripts that are written in general purpose programming

languages (e.g., Pulumi [60]), or when specialized software development kits (e.g., CDK8s [61] and CDKTF [62]) are used. Meanwhile, GLITCH's dataset is the only IaC oracle dataset that provides complete access to individual annotations of the inspectors. Hence, we choose this oracle dataset to conduct our analysis on IaC code smells. Furthermore, it is important to note that GLITCH (and consequently its oracle dataset) is specifically designed to detect the most frequent types of security-related IaC code smells [15]. Hence, we cannot guarantee that our current findings would apply to other less prevalent types of IaC code smells; however, using GLITCH's dataset enables us to study the most encountered types of IaC code smells. We use two previously annotated oracle datasets [15], [53] to triangulate developer perceptions of code smells in IaC and traditional source code. Hence, any limitations faced by these studies directly affect our results. The triangulation is conducted based on Java's object-oriented design. Hence, our results may not apply when IaC code smells are triangulated with other programming paradigms; however, object-oriented programming is one of the most frequently used paradigms and is compatible with most of the popular programming languages [63]. Hence, using Java oracle dataset enable us to position IaC code smells with smells in traditional source code and their most common patterns.

IX. Conclusion

In this paper, we empirically assess developer perceptions of IaC code smells and triangulate their understanding of code smells in IaC and traditional source code. For this purpose, we extensively analyze the oracle dataset of GLITCH, a state-of-the-art detection tool for security code smells in IaC, for the incidences of code smells, their types, and individual annotations of the inspectors of the dataset. Our study reveals that developer perceptions of IaC code smells is considerably impacted by personal bias, leading to annotation disagreements among the inspectors for 46.3% of the oracle issues. On the other hand, the labeling disagreements affect only 1.6% of Java code smells. These results warn developers about the significant differences between their unified understanding of code smells in traditional source code and their misaligned perceptions of IaC code smells. We thereby make recommendations for developers to share their knowledge before annotating IaC code smells to align their ambiguous perceptions of code smells. Furthermore, we advise them to share their annotations in the end, to identify potential cases of annotation negligence towards code smell incidences. Finally, we make recommendations for revising the current catalog of IaC code smells to set guidelines on situations of ambiguity due to the co-occurrences of code smell incidences.

Future Work. We aim to evaluate the extent to which knowledge sharing among developers is beneficial for addressing the threats posed by individual biases. For this purpose, we plan to conduct a controlled experiment involving two groups of participants with one group annotating an oracle dataset individually, while the latter group share their knowledge before and after the annotation. We then compare the extent of uncertainties faced by inspectors in each annotation setting. Annotation uncertainties in the oracle dataset of GLITCH raise concerns about the oracle annotation of similar detection tools. Further investigation of these oracle datasets is necessary to analyze the extent of their annotation uncertainties and influence on the perceived performance of detection tools. Our study reveals the negative effect of current limited taxonomy of IaC code smells on inspector perceptions of them. We plan to address these limitations by integrating hierarchical structure of IaC code examples within the smell definitions—similar to the definition of smells in the object-oriented paradigm. Finally, the conducted evaluation affirms the co-occurrences of admins by default and empty passwords with hard-coded secrets, leading to these smells being confused with each other. It may be possible for other smell types to co-occur as well. To address this issue, we recommend extending the multi-label classification to all co-occurrence incidences concerning varying code smell types. Clearer distinctions among smell types will provide further insights into detection patterns and correlations among smells. We also recommend an evaluation of the impact of these changes on the annotations of oracle datasets by conducting empirical analyses.

X. Acknowledgments

This work is partially supported by the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation alongside of FCT, Fundação para a Ciência e a Tecnologia under grant BD/04736/2023, and projects 2024.07411.IACDC and UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020). We also thank the attendees of the Dagstuhl Seminar 23082 "Resilient Software Configuration and Infrastructure Code Analysis", where the idea for this paper was initially conceived.

References

[1] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *Iet soft.*, vol. 12, no. 3, 2018.
[2] P. Danphitsanuphan and T. Suwantada, "Code smell detecting tool and code smell-structure bug relationship," in *2012 Spring Cong. on Eng. & Tech.*
[3] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of soft.: Evolution & Process*, vol. 27, no. 11, 2015.
[4] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *2019 IEEE/ACM 41st Inter. Conf. on soft. Eng.*
[5] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Trans. on soft. Eng. & Methodology*, vol. 30, no. 1, 2021.
[6] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment." *J. Object Technol.*, vol. 11, no. 2, 2012.
[7] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and a. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *2018 Ieee 25th Inter. Conf. on soft. Analysis, Evolution & ReEng.*
[8] V. Pontillo, D. A. d'Aragona, F. Pecorelli, D. Di Nucci, F. Ferrucci, and F. Palomba, "Machine learning-based test smell detection," *arXiv preprint arXiv:2208.07574*, 2022.
[9] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical soft. Eng.*, vol. 21, 2016.
[10] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, "Deep learning based code smell detection," *IEEE Trans. on soft. Eng.*, vol. 47, no. 9, 2019.
[11] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and a. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 IEEE Inter. Conf. on soft. Maintenance & Evolution*.

[12] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, and B. Xu, "Understanding metric-based detectable smells in python software: A comparative study," *Info. & soft. Tech.*, vol. 94, 2018.

[13] J. De Bleser, D. Di Nucci, and C. De Roover, "Assessing diffusion and perception of test smells in scala projects," in *2019 IEEE/ACM 16th Inter. Conf. on Mining soft. Repositories*, 2019.

[14] K. Morris, *Infrastructure as code: managing servers in the cloud*, 2016.

[15] N. Saavedra and J. F. Ferreira, "Glitch: Automated polyglot security smell detection in infrastructure as code," in *Proceedings of the 2022 37th IEEE/ACM Inter. Conf. on Automated soft. Eng.*

[16] O. Security, "Infrastructure as code: Common risks," https://orca.security/resources/blog/infrastructure-as-code-common-risks/, acc. on: 2024-09-10.

[17] R. Hersher, "Amazon and the $150 million typo," https://shorturl.at/4GZZe, 2017, blog post.

[18] A. A. Elkhail and T. Cerny, "On relating code smells to security vulnerabilities," in *2019 IEEE 5th Inter. Conf. on Big Data Security on Cloud, IEEE Inter. Conf. on High Performance & Smart Computing, & IEEE Inter. Conf. on Intelligent Data & Security*, 2019.

[19] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of Infrastructure-as-Code: Insights from industry," in *2019 IEEE Inter. Conf. on soft. Maintenance & Evolution*.

[20] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 2016 13th Inter. Conf. on Mining soft. Repositories*. Association for Computing Machinery, 2016.

[21] P. R. R. Konala, V. Kumar, and D. Bainbridge, "Sok: Static configuration analysis in infrastructure as code scripts," in *2023 IEEE Inter. Conf. on Cyber Security & Resilience*, 2023.

[22] Y. Brikman, "Why we use terraform and not chef, puppet, ansible, saltstack, or cloudformation."

[23] E. Van der Bent, J. Hage, J. Visser, and G. Gousios, "How good is your puppet? an empirically defined and validated quality model for puppet," in *2018 IEEE 25th Inter. Conf. on soft. Analysis, Evolution & ReEng.*

[24] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Info. & soft. Tech.*, vol. 108, 2019.

[25] 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 Inter. Conf. on soft. Eng.*, 2020.

[26] M. Chiari, M. De Pascalis, and M. Pradella, "Static analysis of infrastructure as code: a survey," in *2022 IEEE 19th Inter. Conf. on soft. Architecture Companion*.

[27] S. K. Mondal, R. Pan, H. D. Kabir, T. Tian, and H.-N. Dai, "Kubernetes in it administration and serverless computing: An empirical study and research challenges," *The Journal of Supercomputing*, 2022.

[28] A. Rahman, D. B. Bose, Y. Zhang, and R. Pandita, "An empirical study of task infections in ansible scripts," *Empirical soft. Eng.*, vol. 29, no. 1, 2024.

[29] 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 & soft.*, vol. 170, 2020.

[30] N. Bessghaier, M. Begoug, C. Mebarki, A. Ouni, M. Sayagh, and M. W. Mkaouer, "On the prevalence, co-occurrence, and impact of infrastructure-as-code smells," in *2024 IEEE Inter. Conf. on soft. Analysis, Evolution & ReEng.*

[31] G.-P. Drosos, T. Sotiropoulos, G. Alexopoulos, D. Mitropoulos, and Z. Su, "When your infrastructure is a buggy program: Understanding faults in infrastructure as code ecosystems," *Proceedings of the ACM on Prog. Languages*, vol. 8, no. OOPSLA2, 2024.

[32] S. Reis, R. Abreu, M. d'Amorim, and D. Fortunato, "Leveraging practitioners' feedback to improve a security linter," in *Proceedings of the 2022 37th IEEE/ACM Inter. Conf. on Automated soft. Eng.*

[33] A. Rahman and L. Williams, "Characterizing defective configuration scripts used for continuous deployment," in *2018 IEEE 11th Inter. Conf. on soft. testing, verification & validation*.

[34] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Within-project defect prediction of infrastructure-as-code using product and process metrics," *IEEE Trans. on soft. Eng.*, vol. 48, no. 6, 2021.

[35] R. M. de Mello, R. Oliveira, and A. Garcia, "On the influence of human factors for identifying code smells: A multi-trial empirical study," in *2017 ACM/IEEE Inter. Symp. on Empirical soft. Eng. & Measurement*.

[36] R. Oliveira, R. de Mello, E. Fernandes, A. Garcia, and C. Lucena, "Collaborative or individual identification of code smells? on the effectiveness of novice and professional developers," *Info & soft. Tech.*, vol. 120, 2020.

[37] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and soft.*, vol. 138, 2018.

[38] N. Saavedra, J. Gonçalves, M. Henriques, J. F. Ferreira, and A. Mendes, "Polyglot code smell detection for infrastructure as code with glitch," in *Proceedings of the 2023 38th IEEE/ACM Inter. Conf. on Automated soft. Eng.*

[39] J. Saldaña, *The coding manual for qualitative researchers*, 2021.

[40] W. G. Cochran, "The combination of estimates from different experiments," *Biometrics*, vol. 10, no. 1, 1954.

[41] R. A. Armstrong, "When to use the bonferroni correction," *Ophthalmic & Physiological Optics*, vol. 34, no. 5, 2014.

[42] Checkov, "Checkov," https://github.com/bridgecrewio/checkov, acc. on: 2024-07-04.

[43] runterrascan, "Terrascan," https://github.com/tenable/terrascan, acc. on: 2024-7-04.

[44] R. A. Fisher, "On the interpretation of χ 2 from contingency tables, and the calculation of p," *Journal of the royal statistical society*, vol. 85, no. 1, 1922.

[45] R. Moussa, G. Guizzo, and F. Sarro, "Meg: Multi-objective ensemble generation for software defect prediction (hop gecco'23)," in *Proceedings of the 2023 Companion Conf. on Genetic & Evolutionary Computation*.

[46] J. Lin, "Divergence measures based on the shannon entropy," *IEEE Trans. on Information Theory*, vol. 37, no. 1, 1991.

[47] S. K. Kachigan, *Multivariate Statistical Analysis: A Conceptual Introduction*, 1986.

[48] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2nd ed., 1988.

[49] M. Sabou, K. Bontcheva, L. Derczynski, and A. Scharl, "Corpus annotation through crowdsourcing: Towards best practice guidelines." in *LREC*, 2014, pp. 859–866.

[50] M. Syed and S. C. Nelson, "Guidelines for establishing reliability when coding narrative data," *Emerging Adulthood*, vol. 3, no. 6, 2015.

[51] A. Sanatinia and G. Noubir, "On github's programming languages," *arXiv preprint arXiv:1603.00431*, 2016.

[52] J. O. Ogala and D. V. Ojie, "Comparative analysis of c, c++, c# and java programming languages," *GSJ*, vol. 8, no. 5, 2020.

[53] L. Madeyski and T. Lewowski, "Mlcq: Industry-relevant code smell data set," in *Proceedings of the 2020 24th Inter. Conf. on Evaluation & Assessment in soft. Eng.*

[54] N. S. Turhan, "Karl pearson's chi-square tests." *Educational Research & Reviews*, vol. 16, no. 9, 2020.

[55] B. Di Eugenio, "On the usage of kappa to evaluate agreement on coding tasks." in *LREC*, vol. 102, 2000.

[56] J. Cohen, "A coefficient of agreement for nominal scales," *Educational & psychological measurement*, vol. 20, no. 1, 1960.

[57] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, 1971.

[58] O. Murphy, "Adoption of infrastructure as code (iac) in real world: Lessons and practices from industry," *Journal of soft. Eng.*, vol. 15, no. 3, 2022.

[59] HashiCorp, "Terraform," https://www.terraform.io/, 2014, acc. on: 2024-07-23.

[60] B. Campbell and B. Campbell, "The aws cdk and pulumi," *The Definitive Guide to AWS Infrastructure Automation: Craft Infrastructure-as-Code Solutions*, 2020.

[61] cdk8s Community, "cdk8s: Cloud development kit for kubernetes," https://cdk8s.io/, 2025, acc. on : 2025-01-26.

[62] HashiCorp, "Cdk for terraform (cdktf)," https://developer.hashicorp.com/terraform/cdktf, 2025, acc. on: 2025-01-26.

[63] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillere, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *IEEE 2013 37th annual computer soft. & appls. Conf.*