

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/377842852>

# On the Prevalence, Co-occurrence, and Impact of Infrastructure-as-Code Smells

Conference Paper · December 2023  
DOI: 10.1109/SANER60148.2024.00009

CITATIONS  
2

READS  
75

6 authors, including:



**Narjes Bessghaier**  
École de Technologie Supérieure

11 PUBLICATIONS 76 CITATIONS


SEE PROFILE



**Mahi Begoug**  
École de Technologie Supérieure

6 PUBLICATIONS 23 CITATIONS


SEE PROFILE



**Ali Ouni**  
École de Technologie Supérieure

186 PUBLICATIONS 5,369 CITATIONS

SEE PROFILE



**Mohammed Sayagh**  
École de Technologie Supérieure

27 PUBLICATIONS 522 CITATIONS

SEE PROFILE

# On the Prevalence, Co-occurrence, and Impact of Infrastructure-as-Code Smells

Narjes Bessghaier\*, Mahi Begoug\*, Chemseddine Mebarki\*, Ali Ouni\*, Mohammed Sayagh\*, Mohamed Wiem Mkaouer†

\*ETS Montreal, University of Quebec, Montreal, QC, Canada

†Rochester Institute of Technology, Rochester, NY, USA

{narjes.bessghaier,mahi.begoug,chemseddine.mebarki}.1@ens.etsmtl.ca, {ali.ouni,mohammed.sayagh}@etsmtl.ca, mwmvse@rit.edu

**Abstract**—In modern software systems, Infrastructure-as-Code (IaC) tools play a pivotal role in automating the management of various infrastructure resources such as networks, databases, and services. This automation is done through code-based specification files, commonly known as IaC files. Similarly to other code files, IaC files can suffer from violations of established implementation and design standards, i.e., *IaC smells*. Although prior research has studied various aspects of traditional smells in non-IaC artifacts, there is little knowledge of how IaC smells are prevalent, co-occurring, and impacting the change and defect proneness of IaC code. To fill this gap, we conduct an empirical study encompassing 82 Puppet-based open-source projects. Our investigation focused on 12 types of IaC smells in both implementation and design levels. Our findings reveal that IaC smells do not manifest uniformly, as IaC smells that are particularly associated with modularity issues, exhibit high prevalence rates across projects. Additionally, we found that 74% of IaC files are smelly and over 52% of the smelly IaC files have at least two co-occurring IaC smells. Furthermore, our findings highlight that, on average, smelly IaC files are modified nearly 3.8 times, in terms of number of commits, more frequently than non-smelly IaC files. Furthermore, smelly IaC files are found to be 3.1 times more prone to larger code changes, in terms of code churn, than non-smelly IaC files. Additionally, we found that smelly IaC files are 3.3 times more prone to the introduction of defects that are likely to persist in 1.65 more commits before being fixed than non-smelly IaC files. These findings advocate developers to be more aware of IaC smells in their projects and consider their correction.

**Index Terms**—Infrastructure-as-Code (IaC), Puppet, Code change, IaC smells, Defects-proneness

## I. INTRODUCTION

Infrastructure-as-Code (IaC) is a modern approach to automatically manage, provision and deploy infrastructure resources, such as networks, databases, and services [1]. Instead of relying on people to manually give instructions to set up the resources, IaC uses specific tools such as Puppet [2], Ansible [3], and Terraform [4], where each of these tools is responsible for automating various aspects of the creation, setting up and configuration of the infrastructure. IaC works by using machine-readable files to consistently automate the management of infrastructures. By using IaC configuration files, developers can ensure consistent deployment of their configurations in diverse environments [5] and apply the same coding practices to the infrastructure as they do with non-IaC artifacts. For instance, the quality of the infrastructure code is maintained through code reviews, version control, and

automated tests [6]. This approach simplifies infrastructure management and scalability while maintaining reliability and predictability, as the system evolves.

However, implementing IaC comes with its own set of challenges [7]. A notable concern is the violation of established best coding practices within IaC files, known as “*code smells*” in traditional software development [8]. We refer to the smells in IaC files as *IaC smells* that manifest as inconsistencies, poor design choices, or violations of standards within the code that defines and configures the underlying infrastructure resources. Given that the management of infrastructure resources can progressively escalate in complexity over time [5, 9], IaC files should be free of smells to avoid potential disruptions [10].

While there exists a large body of research exploring best and poor coding practices in IaC artifacts [11, 12, 13, 14, 15], there exists little knowledge about the prevalence of IaC smells, and the impact associated with these smells on IaC maintenance efforts and IaC defects. For instance, an IaC smell related to deprecated code statements [16] can lead to future compatibility defects. In the project *puppet-swift*<sup>1</sup>, a defect emerged when a deprecated code statement related to the way variables should be accessed was identified<sup>2</sup>. Understanding the prevalence of IaC smells, and their impact on the maintenance efforts of IaC and IaC defect proneness, will raise awareness of the impact of IaC smells and motivate researchers and IaC tool developers to propose solutions for automatically fixing IaC smells.

Therefore, to better understand the impact and prevalence of IaC smells in software projects, we conduct an empirical study on a set of 82 open-source Puppet-based projects<sup>3</sup> and 12 types of IaC smells related to Puppet (an IaC tool). We focus our analysis on Puppet [2] as it is a popular IaC tool in practice, being among the Top-3 IaC tools in terms of use by developers [7] and is widely-used in recent studies [17, 18]. Further, the availability of an established catalog and tool support for Puppet-related smells enables us to conduct a comprehensive analysis [16]. Specifically, we select a catalog of 12 types of IaC smells related to Puppet established by Sharma et al. [16], and analyze their prevalence, co-occurrence

<sup>1</sup><https://github.com/openstack/puppet-swift>

<sup>2</sup><https://bugs.launchpad.net/puppet-swift/+bug/1282717>

<sup>3</sup><https://www.puppet.com>

patterns, and impact on the change- and defect-proneness. To achieve our goal, we address the following research questions.

**RQ1: To what extent are IaC smells prevalent in Puppet-based projects?** Our findings reveal that IaC smells do not manifest uniformly, as smells that are particularly associated with modularity issues, exhibit high prevalence rate.

**RQ2: What types of IaC smells tend to occur together?** Our findings indicate that 74% of the IaC files are smelly and that over 52% of IaC smelly files exhibit the presence of multiple IaC smells, underscoring the prevalence of having poor coding practices within IaC code.

**RQ3: Are smelly IaC files more prone to code changes than non-smelly IaC files?** Our findings highlight that, on average, smelly IaC files are modified, in terms of commits, nearly 3.8 times more frequently than non-smelly IaC files. Furthermore, we found that smelly IaC files are, on average, 3.1 times more prone to code changes (churn) than non-smelly IaC files. This means that identifying and addressing IaC smells could be beneficial in terms of code quality.

**RQ4: Are smelly IaC files more prone to defects than non-smelly IaC files?** We found that smelly files are, on average, 3.3 times more prone to the introduction of defects that are likely to persist in 1.65 more commits before being fixed. Nonetheless, the churn required to fix these defects remains relatively the same for both smelly and non-smelly files. These findings underscore the importance of addressing IaC smells as a proactive measure to mitigate the risk of potential defects that could alter the intended behavior of a Puppet configuration task.

Our paper is organized as follows. Section II summarizes the related work. Section III describes how we formulate the design of our empirical study, while Section IV presents and discusses the results. Finally, Section V discusses our threats to validity, and we conclude and describe our future directions in Section VI.

## II. RELATED WORK

### A. Studies on IaC practices and smells

While there have been numerous recent studies on IaC, most of them have focused on understanding good and bad practices in IaC coding [11], IaC smells detection [12, 13], and IaC quality assurance [10, 14, 15, 16]. Some researchers have focused on the maintainability of IaC code, where maintainability measurement models were proposed for Puppet scripts [14]. Subsequently, Guerriero et al. [5] deepened their understanding of the challenges associated with the development of IaC in industrial settings by conducting a survey involving 44 senior practitioners. They found that IaC practitioners suffer from the lack of IaC debugging tools. In the same vein, Palma et al. [15] provided 46 metrics to evaluate the quality of IaC files. Kumara et al. [11] further established a taxonomy that outlines both best and poor practices about IaC tools such as Ansible, Chef, and Puppet. Their analysis sheds light on the various challenges faced by IaC coding practitioners and the corresponding solutions they employ to mitigate these challenges.

Unlike traditional software systems, researchers investigated the identification of *IaC smells* in IaC files to assess their overall quality. For example, Rahman et al. [12] examined the security aspect of Puppet files by identifying 7 IaC security smells. Schwarz et al. [13] investigated IaC smells for the Chef IaC tool by identifying possible violations of Chef design guidelines. Rian et al. [10] proposed the *Rehearsal* tool to check the non-deterministic Puppet files. Similarly, Sharma et al. [16] proposed the *Puppeteer* tool to detect 13 implementations and 11 design smells. Furthermore, the authors delved into the prevalence and co-occurrence of these smells. Our study expands the work of Sharma et al. [16] to further investigate the prevalence of IaC smells at file and module levels. As well, we employ the *Association rule mining* to examine the extent to which types of IaC smells tend to co-occur.

While the studies mentioned earlier have primarily concentrated on the exploration of best and poor coding practices within IaC, as of our knowledge, there has been no investigation into assessing the level of effort required for maintenance and the defect-proneness arising from the presence of *IaC smells*.

### B. Studies on Code smells prevalence and impact

Unlike IaC, there have been several studies that studied the impact of smells on non-IaC artifacts. For example, Palomba et al. [19] investigated how the prevalence of code smells is related to the size of classes in Java software systems. Their findings highlight that code smells belonging to large and complex code are known to be the most persistent. Similar results are found by Bessghaier et al. [20] when analyzing 12 code smells in the context of PHP web-based projects. Chatzigeorgiou et al. [21] investigated the diffusion of code smells in 24 releases of two Java projects (JFlex, and JFreeChart). The results revealed a growing number of code smell instances as the system progressed.

Other studies investigated the co-occurrence of code smells using the association rule mining algorithm to identify common co-occurring pairs [22]. Palomba et al. [23] provided insights into a large-scale analysis on how co-occurring pairs reside in the system and how developers manage to eliminate coexisting pairs. More recently, Biruk et al. [24] also investigated the coexisting phenomenon between SQL and code smells in Java data-intensive programs. In the same line, Hamdi et al. [25] explored the co-occurrence between 15 types of Android-specific code smells and 10 types of object-oriented code smells in 1,923 Android application. Their results highlight that these are highly co-occurring.

Furthermore, other studies investigated the impact of code smells on the change-proneness of files. Olbrich et al. [26] found that smelly files are more susceptible to code changes and are more likely to experience more change size. An additional study standardizing the size of *God* and *Brain* classes demonstrated that those two classes are less prone to changes. Khomh et al. [27] investigated the impact of different types of smells on the change-proneness and found that the

more instances of smell a class has, the more exposed to changes. These findings were also confirmed by Spadini et al. [28], who found that further code changes result from the existence of test smells, which may lead to faults in the production code.

Moreover, some studies investigated the impact of code smells on the defect-proneness of files. Saboury et al. [29] examined the impact of 12 JavaScript smells on the defect-proneness. The obtained results showed that non-smelly files are less likely to spread defects by 65%. Biruk et al. [24] examined the correlation between traditional code and SQL smells with defects in Java programs. The study indicates that defects are not statistically proven to be correlated with SQL smells, unlike traditional code smells. A broad analytical analysis was performed on over 395 releases of 30 open-source projects by Palomba et al. [19]. The investigation into the effect on the defect-proneness of 13 types of code smells reported that smelly files are 3 times more prone to defects than non-smelly files. Further research indicated that only 21% of defects are introduced before smells are introduced in a class. In addition, classes affected by two or three code smells are likely to witness more defect-fixing activities [19].

In our study, we build upon prior studies to contribute insights into the impact of IaC smells on the maintainability of IaC code and its proneness to defects. We look at examining whether IaC code susceptibility to change and defect are in line with those observed in non-IaC artifacts. To the best of our knowledge, this is the first study that attempted to explore the impact and occurrences of IaC specific code smells.

### III. EMPIRICAL STUDY DESIGN

Our study aims to conduct a comprehensive analysis of IaC smells in Puppet projects, as it is one of the most popular IaC tools, focusing on three main aspects: (1) *Prevalence*, which assesses the occurrence frequency of IaC smells within IaC files; (2) *Co-occurrence* that investigates the patterns of IaC smells that commonly co-occur within the same file; and (3) *Impact*, which entails evaluating how IaC smells affect the change- and defects-proneness of IaC files. We make all scripts and data collected during this study available for future extensions and replications on IaC smells [30]. Figure 1 presents an overview of our empirical study composed of 3 main steps, (Step A) projects selection, (Step B) IaC smells detection, and (Step C) data analysis.

#### A. Projects selection

To prepare our dataset, we used the GitHub API<sup>4</sup> to systematically retrieve projects created since 2012 that employ the programming language “*puppet*”. Initially, this selection resulted in a preliminary set containing 22,177 projects. To refine our dataset, we filtered out forked, private, and archived projects. Specifically, we retained projects with a balance of star count of at least 5 and a minimum of 80 commits, ultimately reducing the list to 414 projects. Next, we examined the projects to exclude any that incorporate other IaC

tools besides Puppet. The rationale behind this decision is to maintain a focused analysis of the changes specific to a single IaC tool (Puppet in our study). The presence of files related to multiple IaC tools within one project could potentially lead to an inflation of change-proneness for one of the IaC tools, which might distort our findings. By curating the dataset in this manner, we aim to mitigate the risk of overestimating the impact of changes on files associated with other IaC tools than Puppet. Thus, we ensure a more accurate assessment of the target IaC tool’s change-proneness. We ended up identifying 6 projects with IaC files associated with the Ansible IaC tool, 1 project with Chef-related files, and 95 projects with IaC files linked to Terraform. Following this step, we were left with a subset of 102 projects that exclusively pertained to Puppet. To ensure the dataset’s quality, we further excluded any private projects that were not accessible by the GitHub API, resulting in a final selection of 82 projects.

#### B. IaC smells detection

To collect our dataset of IaC smells occurring within Puppet IaC files with “.pp” extension, we considered a comprehensive set of 12 types of IaC smells associated with both implementation and design aspects established by Sharma et al. [16]. We used the “*Puppeteer*” tool [16] to identify instances of IaC smells related to design issues. In addition, we used the “*Puppet-Lint*” tool version that was extended by Sharma et al. [16] to detect instances of IaC smells related to implementation issues. We provide a description of the 4 implementation and 8 design-related IaC smells that have been identified in our projects, as outlined in Table II. It is important to note that the IaC smells are identified at two levels of abstraction, (1) at the *file level*, we find all instances of IaC smells within individual files, and (2) at the *module level*, we find IaC smells instances that emerge when interactions occur between IaC files within the same module or between modules using the “*include*” command<sup>5</sup>. We provide in Table I the statistics of the studied projects.

TABLE I. The studied projects’ statistics.

Statistic	Count
Total number of projects	82
Total number of commits	19,641
Total number of smelly IaC files	1,462
Total number of non-smelly IaC files	501
Total number of smells instances	5,213
Studied period	2011-2023
Count of stars	5-79

#### C. Data Analysis

In this subsection, we describe the analysis method we use to address each of our defined research questions.

**IaC smells prevalence (RQ1):** We consider a total of 12 Puppet-related IaC smells that pertain to both implementation and design aspects. For each IaC file, denoted as  $F_i$ , within a given project  $P_j$ , we calculate the frequency of occurrences

<sup>4</sup><https://docs.github.com/en/rest?apiVersion=2022-11-28>

<sup>5</sup><https://subscription.packtpub.com/book/cloud-and-networking/9781784394882/1/ch011v11sec16/using-modules>

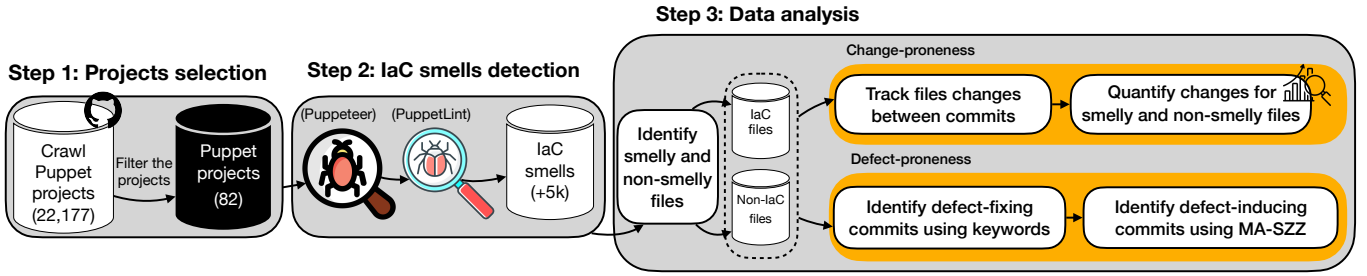


Fig. 1: Overview of our empirical study

TABLE II. List of IaC implementation and design smells in our study.

IaC smells	Description	Level	Artifact
Complex Expression (ICE)	It occurs when the code contains a complex expression that is difficult to understand [16].	Implementation	File
Incomplete Tasks (IIT)	It occurs when the code contains <code>fix me</code> and <code>to-do</code> tags indicating incomplete tasks [16].	Implementation	File
Deprecated Statement Usage (IDS)	It occurs when the code uses one of the obsolete instructions (such as <code>import</code> ) [16].	Implementation	File
Incomplete Conditional (IIC)	It occurs when a conditional structure <code>if...elseif</code> is used without a final <code>else</code> clause [16].	Implementation	File
Multifaceted Abstraction (DMF)	It occurs when each abstraction (e.g., a resource, a class, or a module) does not follow the principle of single responsibility [16].	Design	File
Insufficient Modularization (DIM)	It occurs when an abstraction is large or complex and could, therefore, be further modularized [16].	Design	File
Tightly Coupled Modularization (DTC)	It occurs when two modules are tightly coupled [16].	Design	File
Weakened Modularity (DWM)	It occurs when a module exhibits high coupling and low cohesion [16].	Design	File
Unstructured Module (DUM)	It occurs when a module does not have a well-defined and coherent structure [16].	Design	Module
Missing Dependency (DMP)	It occurs when there are missing dependencies between the declared classes and the included classes [16].	Design	Module
Dense Structure (DDS)	It occurs when the code contains excessive and dense dependencies without any particular structure [16].	Design	Module
Deficient Encapsulation (DDE)	It occurs when the definition of a node declares a set of global variables to be retrieved by the classes included in the definition [16].	Design	Module

for each of the 12 IaC smell types. Furthermore, we evaluate the prevalence of these IaC smells per thousand code lines (KLOC). The KLOC-based evaluation allows us to contextualize the prevalence of IaC smells relative to the size of the files, providing a more meaningful perspective on their distribution.

**IaC smells co-occurrence (RQ2):** We employ the Apriori association-rule mining algorithm [31], a well-established method in association rule mining, used to unveil frequent item sets within our dataset, which in our case, are pairs of IaC smells. The output of the Apriori algorithm consists of a collection of association rules, each signifying associations between pairs of IaC smells in our study. To quantify the strength of association for each rule (*i.e.*, a specific pair of IaC smells), we employ a set of three distinct measures, *Support* [32], *Confidence* [32], and *Lift* [33].

- **Support:** it explores the extent to which a pair of IaC smells ( $ST_1$  and  $ST_2$ ) tend to occur together within the same commit.

$$\text{Support}(ST_1 \Rightarrow ST_2) = \frac{ST_1 \cup ST_2}{\text{Transactions}} \in \{0 - 1\} \quad (1)$$

- **Confidence:** it quantifies the probability that an IaC smell is associated with the presence of another IaC smell, and its values range between 0 and 1.

$$\text{Confidence}(ST_1 \Rightarrow ST_2) = \frac{\text{Support}(ST_1 \Rightarrow ST_2)}{\text{Support}(ST_1)} \in \{0 - 1\} \quad (2)$$

- **Lift:** it explores the ratio of dependence between IaC smells. Its value range spans from 0 to  $+\infty$ . When the lift

value surpasses 1, it indicates a high correlation between the pair of IaC smells, suggesting a stronger likelihood of a causal relationship.

$$\text{Lift}(ST_1 \Rightarrow ST_2) : \frac{\text{Support}(ST_1 \Rightarrow ST_2)}{\text{Support}(ST_1) * \text{Support}(ST_2)} \text{ in } \{0 - 1\} \quad (3)$$

To mitigate potential bias against infrequently occurring IaC smells, we adopt the approach of Bessghaier et al. [20], entailing the calibration of the *support* threshold, with the objective of encompassing 80% of the observed IaC smell pairs within our analysis. Additionally, we fixed the *Lift* value to 1, reflecting our emphasis on identifying strongly correlated instances of IaC smells.

Furthermore, we employ the Chi-squared [34] test and Cramer's V test [35] tests to gauge the degree of association between the IaC smells in our study. The Chi-squared test is a statistical method used to evaluate the significance of variation within the same population between two categorical variables (in our context, the combined dataset of all projects). This test is designed to assess the null hypothesis, denoted as  $-H_{20}$ : *IaC smells occur independently of each other*. The Cramer's V, on the other hand, is an effect size measure applied alongside the Chi-squared Pearson test to quantify the strength of the relationship between variables.

**IaC smells impact on change-proneness (RQ3):** To study the impact of IaC smells on the change-proneness, we mine the change history of each project using the `git` versioning system. We track all changed Puppet files (with the “`pp`”

extension) in all commits. Then, using the Puppeteer and Puppet-Lint tools, we determine whether the retrieved files in every single commit fall into the category of being either smelly or non-smelly. Following this classification, we use the “*Change-frequency*” metric to assess the frequency with which files containing IaC smells undergo modifications in comparison to non-smelly files. This involves quantifying the number of commits in which a modified IaC file, denoted as  $F$ , experiences updates between two consecutive commits. Moreover, we proceed to compute the change-proneness of a modified IaC file, using Equation 4. This measure quantifies the extent of changes made to the file across all commits  $com_j$  in a project. This analysis aims to test the null hypothesis:  $-H_{30}$  : *smelly IaC files are not more likely to undergo modifications than non-smelly IaC files*.

$$\text{Change-proneness}(F, com_i) = \sum_{i=1}^{i=n} \text{churn}(F, com_i) \quad (4)$$

where  $n$  represents the total number of commits. The function  $\text{churn}(F, com_i)$  calculates the code churn, which corresponds to the sum of the added and removed code lines in the IaC file  $F$  in the commit  $com_i$  using The PyDriller framework [36].

**IaC smells impact on defect-proneness (RQ4):** This research question focuses on examining the introduction of defects during Puppet software development. To achieve this goal, we analyzed the “*Defect-Fixing Commits (DFC)*” and “*Defect-Inducing Commits (DIC)*” and their correlation with IaC smells. We define a defect-fixing commit as a code commit to fix defects, while a defect-inducing commit is a code commit where a defect is first introduced [37]. To extract the DICs, we take advantage of the SZZ algorithm proposed by Sliwerski et al. [38]. The SZZ approach can be divided into two primary steps: the identification of defect-fixing commits (DFCs) and the identification of defect-inducing commits (DICs).

1- *DFCs identification*: To identify defect-fixing commits, we initiate the process by selecting commits from each project that involve modifications to at least one IaC file. Subsequently, for each of these commits, we examine their commit messages. If a commit message contains any of the specific keywords previously employed by established studies for defect identification [39, 40], such as “error”, “bug”, “fix”, and “issue”, we categorize it as a defect-fixing commit.

2- *DICs identification*: Subsequently, to identify the list of defect-inducing commits, we apply Meta-change Aware SZZ [41] as a variant of SZZ, which is recommended by Fan et al. [42]. In their study, Fan et al.[42] explored the consequences of inaccurately labeled defect-related changes detected by the SZZ algorithm. They examined the results generated by four distinct SZZ variants: B-SZZ [41], AG-SZZ [43], MA-SZZ [42], and RA-SZZ [44]. Notably, their findings led them to assert that the mislabeled changes produced by MA-SZZ did not lead to wasted inspection effort in terms of LOC.

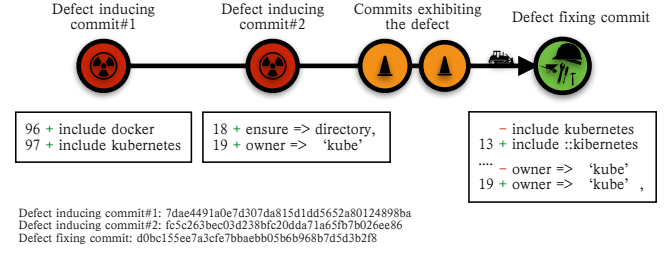


Fig. 2: MA-SZZ detection process of defects inducing commits.

Given the defect-fixing commit, MA-SZZ (1) searches for all IaC files that have been modified. Then, for each modified IaC file in the DFC, the MA-SZZ (2) applies the *diff* command to determine the modified (undergone the fixing change) lines of code ( $loc_{i=1}^i \rightarrow n$ ). Afterward, for each modified line of code  $loc_i$  in the DFC, the MA-SZZ (3) employs the *blame* git command to determine the commits (“*Commits-Exhibiting Defects (CED)*”) where  $loc_i$  was modified in previous versions of the code (previous IaC file revisions), and thus, identifies the last change (*i.e.*, the DIC) that first changed the modified  $loc_i$ . An example of the MA-SZZ process is illustrated in Figure 2, where we identified the DFC with the keyword “*Fix*”. In this DFC, we have the “*node.pp*”<sup>6</sup> IaC file that has been modified with an example of two modified line indexes {13, 19}. Employing the MA-SZZ algorithm, we have identified two commits as Defect-Inducing Commits (DIC). In the first DIC (Commit #1), a new line of code was introduced with an index of 97. In the second DIC (Commit #2), the code line was added with index 19. It is noteworthy that both of these added lines, indexed as {97, 19}, were subsequently rectified in the DFC. Therefore, we identify the initial commits that introduced changes to these lines of code as DIC. The commits that fall in between the DIC and the DFC, where the file is also modified, are categorized as Commits-Exhibiting Defects (CED). This analysis aims to test the null hypothesis:  $-H_{40}$  : *smelly IaC files are not more prone to defects than non-smelly IaC files*.

To validate the precision of the identified defect-fixing commits and defect-inducing commits, we performed a manual evaluation of a randomly selected sample consisting of 375 commits, chosen based on a 95% confidence interval. In this evaluation, We thoroughly examined whether the defect-fixing commits addressed defects and whether the defect-inducing commits were, indeed, linked to the changes made in the corresponding defect-fixing commits. The first two authors, having more than 6 years of experience in software engineering, including 3 years of expertise in Infrastructure-as-Code (IaC), collaborated on this assessment, categorizing the commits into one of three classifications: “True”, “False” or “Unclear”. Any uncertainties between the two authors were resolved through discussions until a consensus was reached. Following this manual review, we computed the precision of

<sup>6</sup><https://github.com/cristifalcas/puppet-kubernetes/blob/master/manifests/node.pp>



commits labeled as either “True” or “False”. The precision rate for defect-fixing commits was determined to be 85.6%. As for the defect-inducing commits, the precision rate was 97.6%.

Upon gathering all the necessary data, we used statistical analysis to evaluate the significant difference between our null hypotheses  $H_{30}$  and  $H_{40}$ . Thus, we employed the non-parametric Mann-Whitney U-test [45] with a confidence level of 95% (significance level of  $p\text{-value} < 0.05$ ). The Mann-Whitney U-test assesses the likelihood of one group displaying dominance over the other group. To provide a comprehensive analysis, we complement this test by measuring the non-parametric effect size of the Cliff delta [46]. The effect sizes are classified as negligible for  $d < 0.147$ , small for  $d < 0.33$ , medium for  $d < 0.474$ , and large for  $d \geq 0.47$ . It is worth noting that a file is considered to be smelly if it has at least one code smell instance. Moreover, we conduct an analysis to assess the correlation between the size of files changes (Churn) and the number of lines of code (LOC) to investigate how the LOC measure influences the extent of files changes. This investigation was carried out using the Kendall correlation test [47]. To go deeper into this analysis, we organized the file sizes into three groups based on the LOC distribution across all projects using boxplots quartiles, small [ $\leq 38$  LOC], medium [ $> 38$  AND  $\leq 147$ ], and large [ $> 147$ ].

#### IV. RESULTS ANALYSIS

##### A. RQ1: IaC smells prevalence

**Motivation:** This research question aims to provide insights into the types of IaC smells that are present across the projects and their frequency of occurrence within IaC files.

**Results:** *IaC smells do not manifest uniformly, and they exhibit varying levels of prevalence (i.e., smells diffused across the projects) and frequency (i.e., occurrence number within a file), falling into three distinct patterns.* We report in Figure 3, (a) the average number of IaC smells at file and module levels, (b) the average number of affected files by each type of IaC smells at the file level, (c) the density of IaC smells per KLOC at the file level. Furthermore, in Table III, we provide statistics encompassing the percentages of affected projects, the distribution of IaC smells within smelly IaC files and modules, and the overall percentage of IaC smells. Upon our analysis, we notice the presence of three distribution patterns of IaC smells, characterized by their prevalence and frequency: (1) a high prevalence and high frequency, (2) a high prevalence and low frequency, and (3) low prevalence and low frequency.

**High prevalence and high frequency:** Within this category, we identify the subset of IaC smells that exhibit a high presence across the majority of projects with a high occurrence in IaC files. At the *file level*, the “*Insufficient Modularization (DIM)*” smell is found in all projects (100%) and makes up about 30.57% of all IaC smells we have identified. Specifically, this smell affects nearly 56% of smelly IaC files, with an average of about 18.75 affected IaC files per project and a total percentage of 55.92% smelly IaC files across the projects. This points to a common problem across all projects, where

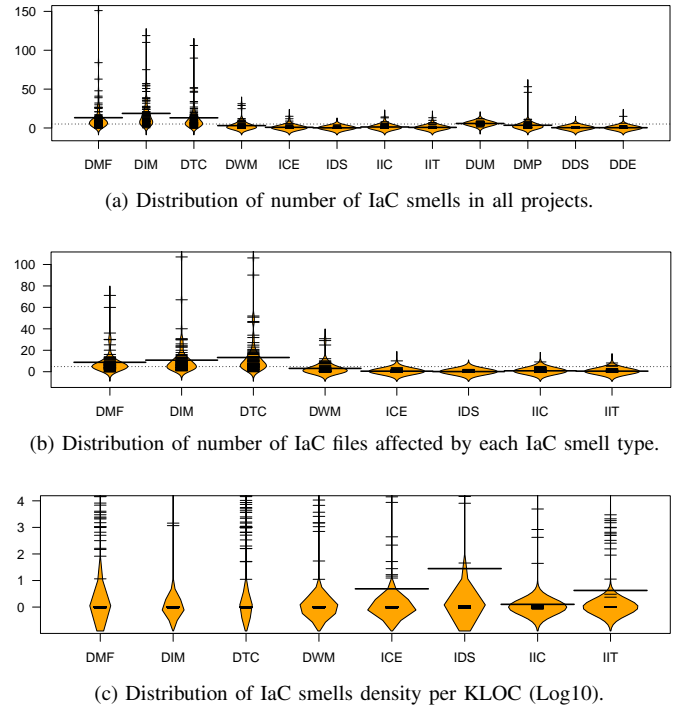


Fig. 3: Distributions graphs depicting the prevalence of IaC smells in the studied projects.

Puppet IaC files are either too complex or exceed the 40-line code limit [16]. This suggests the need for improvement in terms of simplifying these IaC files. On average, we find about 17 instances of the “*Insufficient Modularization (DIM)*” IaC smell for every thousand lines of code (KLOC). The project “*Puppet-Brocade-vTM*” has the highest number of occurrences, with 119 instances. Another prevalent smell is the “*Tightly Coupled Modules (DTC)*”, affecting 96.42% of projects and showing up in 68.70% of the IaC smelly files, contributing to 21.35% of all IaC smells. In one project, it is likely to have an average of 13 coupled modules. We also find the “*Multifaceted Abstraction (DMF)*” smell, making up 21.59% of all identified smells and impacting a significant number of projects (95.23%) and about 45.55% of smelly IaC files. At the *module level*, we observe that the “*Unstructured Module (DUM)*” smell is spread across all analyzed projects (100%) and is present in 97.60% of the modules. This demonstrates that in all projects, there is at least an average of 5.8 of modules that do not adhere to the Puppet pre-defined structure [16]. The projects most affected by this smell are “*puppet-monit*” and “*puppetlabs-patching\_as\_code*” each with 12 occurrences.

**High prevalence and low frequency:** This category encompasses IaC smells that are widespread across the majority of projects but exhibit a relatively lower frequency within individual IaC files. At the *file level*, the only IaC smell falling into this category is the “*Weakened Modularity (DWM)*” smell, which manifests when an IaC file demonstrates high coupling and low cohesion. This particular IaC smell is

identified in 59.52% of the projects and is observed in 16.04% of the smelly IaC files, constituting a mere 5% of the overall number of IaC smells. Notably, the projects most impacted by this smell include “*puppet-icinga2-legacy*” with a total of 31 occurrences, followed by “*cesnet-hadoop*” with 29 instances and “*ankus-modules*” with 25 occurrences. These findings suggest that a high number of IaC files within these three projects exhibit a modularity ratio below 1 [16]. Collectively, these three projects account for 85 occurrences out of the total 261 instances of this smell, amounting to 32.56% of its prevalence. At the *module level*, the only smell in this category is the “*Missing Dependency (DMP)*” which is present in 20.55% of the modules and affects 75.29% of the projects.

**Low prevalence and low frequency:** This category contains IaC smells that are present in a few projects and affect a limited number of IaC files. At the *file level*, all the IaC smells in this category only represent 5.06% of the total number of IaC smells affecting IaC files. Particularly, these IaC smells are all related to implementation violations. For instance, the IaC smell with the lowest prevalence and frequency is the “*Deprecated Statement (IDS)*” smell, which only affects 0.74% of the total smelly IaC files and 10.71% of the studied projects. It is also the least recurrent smell, accounting for only 0.32% of the total detected smells. At the *module level*, two IaC smells were detected in this category; “*Dense Structure (DDS)*” and “*Deficient Encapsulation (DDE)*”. The “*Dense Structure (DDS)*” smell is present in 32.94% of the projects and is detected when a module has excessive and complex dependencies on other modules. The “*Deficient Encapsulation (DDE)*” smell has a spread of 9.41%, indicating that the affected projects suffer from poor use of global variables, making them accessible from and usable by any part of the code.

In summary, our analysis has uncovered a total of 4 implementation-related IaC smells and 8 design-related IaC smells distributed across all the projects we examined. Among these, the most prevalent IaC smells, impacting a substantial number of IaC files, are “*Multifaceted Abstraction*”, “*Insufficient Modularization*”, “*Tightly Coupled Modules*”, and “*Unstructured Module*” all of which belong to the category of design smells. These 4 smells primarily pertain to concerns related to code complexity, length, and the allocation of unique responsibilities. Conversely, implementation smells, although fewer, are not widely dispersed and appear to be limited to specific projects.

#### B. RQ2: IaC smells co-occurrence

**Motivation:** Understanding how frequently IaC smells tend to co-occur is of importance, as it can reveal potential patterns and relationships between different IaC smells. This knowledge can inform developers about common coding issues that often appear together, enabling them to proactively address multiple problems when they encounter one.

**Results:** *Multiple types of IaC smells can concurrently affect various code components of IaC files, with over 52.29% of these IaC files containing more than one type of IaC*

TABLE III. Prevalence and frequency of IaC smells in the analyzed projects at the file and module levels.

IaC smells	% in projects	% in smelly IaC files/modules	% of IaC smells
<b>File level IaC smells</b>			
High prevalence and high frequency			
DIM	100%	55.92%	30.57%
DTC	96.42%	68.70%	21.35%
DMF	95.23%	45.55%	21.59%
High prevalence and low frequency			
DWM	59.52%	16.04%	5%
Low prevalence and low frequency			
IIC	34.52%	4.38%	2.09%
ICE	25%	2.59%	1.42%
IIT	23.8%	2.77%	1.23%
IDS	10.71%	0.74%	0.32%
<b>Module level IaC smells</b>			
High prevalence and high frequency			
DUM	100%	97.60%	9.44%
High prevalence and low frequency			
DMP	75.29%	20.55%	5.73%
Low prevalence and low frequency			
DDS	32.94%	6.58%	0.63%
DDE	9.41%	1.59%	0.59%

**smells.** Previous investigations by Palomba et al. [19, 23] in the context of Android mobile applications and Bessghaier et al. [20, 48] in the domain of web applications have demonstrated that over 60% of source code is affected by multiple code smells. Our study extends these findings by verifying the frequency of IaC smells co-occurrence within Puppet projects, and we found that while 47.71% of IaC files exhibit a single type of IaC smell, a significant 52.29% of the IaC files contain two or more distinct IaC smells. Specifically, we have 28.77% of IaC files having 2 IaC smells, 18.85% with 3 IaC smells, up to 4.67% with 4 types of IaC smells. Prior research has indicated that the simultaneous presence of smells can amplify the susceptibility of code to changes [19, 20, 48]. Therefore, these findings reflect the importance of understanding how our analyzed IaC smells consolidate the co-occurrence phenomenon and its effect on source code changes.

We delve into the co-occurrence patterns of IaC smells within code components, exploring whether the presence of one type of IaC smell implies the presence of another. In Table IV, we present the Top-10 pairs of IaC smells that exhibit strong associations based on their lift values across all the projects we studied. Notably, the pair consisting of the {“*Deficient Encapsulation (DDE)*” → “*Dense Structure (DDS)*”} IaC smells stands out with a high lift value of 40.17 and a confidence level of 0.63. For instance, within the context of the “*cesnet-hadoop*” project, we have identified a noteworthy prevalence of the “*Deficient Encapsulation (DDE)*” code smell, manifesting itself in 15 distinct instances. The “*Deficient encapsulation (DDE)*” smell is indicative of a poor coding practice where modules excessively employ ‘`include`’ statements, a phenomenon associated with the heavy reuse of global variables throughout the codebase. Furthermore, we have also observed an occurrence of the “*Dense Structure (DDS)*” smell, which is characterized by the presence of at least two closely interconnected modules within the project. This observation highlights the interconnectedness and



TABLE IV. Top-10 pairs of IaC smells based on the Lift value for all projects combined.

IaC smells pairs	Support	Confidence	Lift
DDE⇒DDS	<0.01	0.63	40.17
DDS⇒DMP	0.01	0.67	13.73
DDE⇒DMP	<0.01	0.5	10.3
IDS⇒ICE	<0.01	0.17	8.42
DDE⇒DUM	<0.01	1	4.34
ICE⇒IIC	<0.01	0.14	4.27
DUM⇒DMP	0.044	0.19	3.96
DDS⇒DUM	0.013	0.85	3.68
DWM⇒DMF	0.071	0.58	1.67
DMF⇒DIM	0.236	0.68	1.59

complexity of the codebase. Similarly, the pair composed of {“Dense structure (DDS)” → “Missing dependency (DMP)”} IaC smells exhibits a high lift value of 13.73 and a confidence level of 0.67, indicative of a strong association between these IaC smell pairs. Additionally, our analysis revealed a confidence value of 1 for the pair {“Deficient encapsulation (DDE)” → “Unstructured module (DUM)”}, implying that a majority of the Puppet projects lack a predefined and organized structure, particularly in the way they handle different modules and their respective manifests. These unstructured modules exhibit a notable deficiency in encapsulation, meaning that they lack a clear and well-defined boundary between different components. In these cases, modules are often not structured, and they heavily rely on the ‘‘include’’ statement to interconnect various code components. Therefore, the presence of the “Deficient encapsulation (DDE)” IaC smell could be an indicator of the presence of the “Unstructured module (DUM)” IaC smell within the code.

To comprehensively investigate the relationships between pairs of IaC smells, we employed both the Chi-squared test [34] and Cramer’s V test [35] to determine their statistical significance. Table V provides the outcomes of our Chi-squared test, assessing the null hypothesis  $H_{20}$ , assuming complete independence between two IaC smells, essentially investigating the causality between co-occurring IaC smells. Among the top-10 IaC smells pairs examined in Table V, all pairs with p-values <0.05 led us to reject the null hypothesis (p-values are highlighted in bold). Further exploration of Cramer’s V scores reveals the degree of association among these pairs of IaC smells. Notably, the three IaC pairs {“Multifaceted Abstraction (DMF)” → “Insufficient Modularization (DIM)”}, {“Unstructured Module (DUM)” → “Missing Dependency (DMP)”}, and {“Missing Dependency (DMP)” → “Dense Structure (DDS)”} demonstrates a moderate association, yielding a Cramer’s V score over 0.35. All other IaC smells pairs display a weak association, with Cramer’s V scores ranging less than 0.27. It is worth noting that a high co-occurrence rate does not necessarily imply a high degree of association. For instance, while “Deficient Encapsulation (DDE)” exhibits a strong co-occurrence with “Weakened Modularity (DWM)”, indicated by a perfect confidence value in Table IV, it possesses a weak degree of association, as reflected by a Cramer’s V score of 0.1. This underscores the importance of considering both co-occurrence

TABLE V. Top-10 most correlated IaC smells pairs based on Chi-square and Cramer’s V tests.

IaC smells pairs	Chi2 p-value	Cramers_v
DMF⇒DIM	<0.01	0.37
DUM⇒DMP	<0.01	0.36
DMP⇒DDS	<0.01	0.35
DDS⇒DDE	<0.01	0.27
DMF⇒DWM	<0.01	0.18
DUM⇒DDS	<0.01	0.18
DMP⇒DDE	<0.01	0.11
DUM⇒DDE	<0.01	0.1
ICE⇒IIC	<0.01	0.08
ICE⇒IDS	<0.01	0.06

and association metrics when assessing the relationships between IaC smells.

### C. RQ3: IaC smells impact on change-proneness

**Motivation:** Understanding how IaC smells influence the extent of changes (*i.e.*, churn) made to IaC files, can contribute to more maintainable and of improved quality infrastructure configurations by fostering the need to rectify these IaC smells.

**Results:** *Smelly IaC files are 3.8 times more likely to be modified, in terms of commits, compared to non-smelly IaC files, with 3.1 times more prone to code changes (churn).* In Figure 4, we observe that smelly IaC files are prone to frequent changes compared to non-smelly IaC files for small, medium, and large sized IaC files. Overall, smelly IaC files are almost 3.8 more subject to be modified, with an average of 12.44 times per file, than non-smelly IaC files, with an average of 3.24 times per file, along with the project’s evolution. To statistically assess the significance of the difference, we supported this result by computing the Mann-Whitney and Cliff’s delta effect size. The Mann-Whitney proves the significant difference with a p-value < 0.01 and a large effect size of 0.577. Precisely, the small sized smelly IaC files are 3.7 times more prone to changes than non-smelly small IaC files with a p-value < 0.01 and a large effect size of 0.647. The medium-sized smelly IaC files are 3.6 times more prone to changes than medium-sized non-smelly IaC files with a p-value < 0.01 and a large effect size of 0.645. Lastly, the large smelly IaC files are 7.2 more prone to changes than large non-smelly IaC files with a p-value < 0.01 and a large effect size of 0.776.

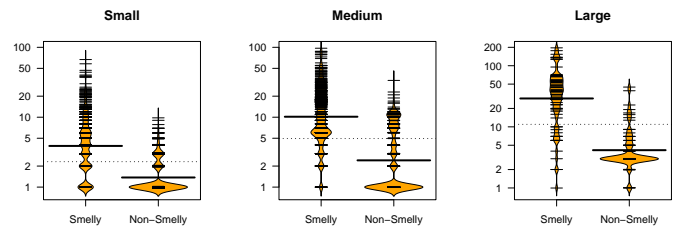


Fig. 4: Distribution of the change frequency of smelly and non-smelly IaC files for small, medium, and large sized IaC files.

To further examine the maintenance effort, we report in Figure 5 the spectrum of code change sizes, *i.e.*, code churn, in both smelly and non-smelly for small, medium, and large

IaC files. We observe from Figure 5 that smelly IaC files have a higher code change size than non-smelly IaC files. These findings indicate that code fragments affected by IaC smells may require increased maintenance efforts, as they are more prone to code change. Specifically, the average code change size that smelly IaC files could experience along the project’s evolution is 232 with a median of 91, whereas non-smelly IaC files experience an average code change size of only 76 with a median of 28. Statistically, the Mann-Whitney test has shown a significant difference between the two populations with a p-value  $< 0.01$  and a medium effect size of 0.446, giving us statistical evidence to reject the null hypothesis  $H_{30}$ . On average, smelly IaC files are almost 3.1 times more subject to code changes than non-smelly IaC files. Similarly to previous studies in Java object-oriented systems [19, 49] and web applications [20, 48], we witnessed similar findings reported in Figure 5, where smelly IaC files exhibit a higher level of proneness to change compared to non-smelly IaC files. The smelly IaC files have their maintenance requirements, which tend to be related to the coexistence of various types of IaC smells, such as “*Complex Expression (ICE)*” and “*Dense Structure(DDS)*”.

Precisely, in small, medium, and large sized IaC files, we find that smelly IaC files are always exhibiting a higher code change compared to non-smelly IaC files. Small sized smelly IaC files are prone to undergo an average of 77 churn and a median of 44, compared to an average of 34 and a median of 12 for small non-smelly IaC files with a p-value  $< 0.01$  and a large effect size of 0.601. Medium size smelly IaC files are prone to an average of 265 code changes and a median of 185, compared to an average of 76 and a median of 52 in non-smelly medium-sized IaC files with a p-value  $< 0.01$  and a large effect size of 0.613. The large sized smelly IaC files are prone to an average of 1405 and a median of 898 code changes, compared to only an average of 264 and a median of 138 in non-smelly IaC files with a p-value  $< 0.01$  and a large effect size of 0.869.

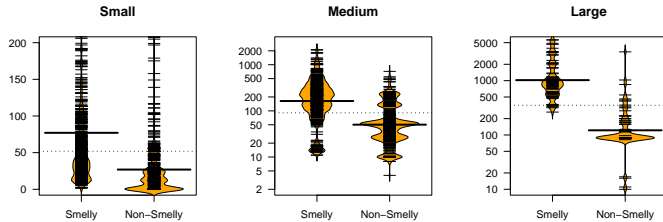


Fig. 5: Distribution of code changes (churn) of smelly and non-smelly IaC files for small, medium, and large sized IaC files.

#### D. RQ4: IaC smells impact on defect-proneness

**Motivation:** By understanding the relationship between IaC smells and defects, developers would prioritize rectifying IaC smells and focus their efforts on reviewing and testing files with IaC smells.

**Results:** *Smelly IaC files are 3.3 times more prone to the introduction of defects that are likely to persist in 1.65 more commits before being fixed than non-smelly IaC files. Nonetheless, the churn required to fix these defects remains relatively the same for both smelly and non-smelly IaC files.*

To analyze the impact of IaC smells on defect proneness, we first investigate the number of defect-inducing commits (DIC) in smelly and non-smelly IaC files for small, medium, and large sized. Smelly IaC files are found to be 3.3 times more prone to defects than non-smelly IaC files with a significant p-value  $< 0.01$  with a medium effect size. Specifically, as shown in Figure 6, a consistent trend emerges across varying IaC file sizes, where smelly IaC files exhibit a higher number of induced defects. Specifically, within the subset of small IaC files, smelly IaC files exhibit an average of 0.79 DIC, in contrast to 0.55 for non-smelly IaC files, revealing a statistically significant difference with a p-value  $< 0.01$  and a small effect size. The medium-smelly IaC files witness an even higher difference with an average of 1.41 compared to 0.36 for the non-smelly IaC files, with a significant p-value  $< 0.01$  and a large effect size. Typically, large smelly IaC files are more prone to defects with an average of 1.94 compared to 0.22, with a significant p-value  $< 0.01$  and a large effect size. As highlighted in previous studies [19, 20, 29] in different programming languages, smelly IaC files are more prone to defects. The reason is that IaC smells represent poor coding practices that could lead to defects if an IaC file is carelessly modified. For example, the “*Incomplete Conditional (IIC)*” smell represents incomplete or improperly structured conditionals that can result in configurations that do not respond correctly to various scenarios. For instance, if conditional statements are not adequately constructed in Puppet IaC files, the system may not behave as expected under certain conditions, such as the example of the “*cli.pp*” file<sup>7</sup> of the project “*puppet-wp*”<sup>8</sup>, where some closing ‘*else*’ branches are missing.

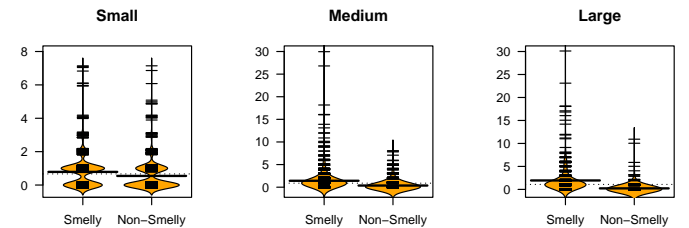


Fig. 6: Distribution of defect-inducing commits in smelly and non-smelly IaC files for small, medium, and large sized IaC files.

To investigate the propagation of defects in our studied projects, we count the CED between the DIC and the DFC. In general, we found that defects persist in smelly IaC files in 1.65 more commits than non-smelly IaC files, with significant difference of p-value  $< 0.01$  and medium effect size. As

<sup>7</sup><https://github.com/Chassis/puppet-wp/blob/master/manifests/cli.pp>

<sup>8</sup><https://github.com/Chassis/puppet-wp>

reported in Figure 7, we observe that small and medium smelly IaC files have a slightly higher average of 4.2 and 8.6 compared to 3.8 and 7.3 with non-significant p-values equal to 0.05 and 0.07, respectively, with negligible effect sizes. However, the large smelly IaC files are found to undergo a higher number of DEC before a defect gets fixed in the DFC with an average of 18 CEDs compared to 10.5 CEDs for the non-smelly IaC files, with a significant p-value of 0.04 and small effect size. For example, the “*agent.pp*” large file<sup>9</sup> of the “*puppet-zabbix*”<sup>10</sup> project took 22 CEDs to fix a legacy fact that was explicitly commented as “*# the agent doesn't work perfectly fine with selinux.*”

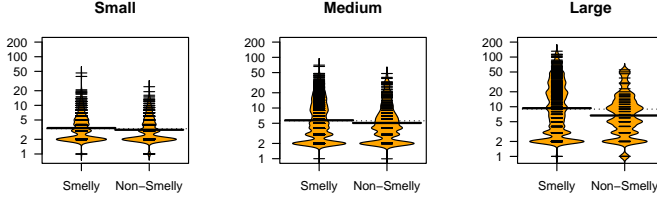


Fig. 7: Distribution of defect-exhibiting commits of smelly and non-smelly files for small, medium, and large sized files.

Finally, to get a deeper understanding of the defect-proneness phenomenon, we investigate the required code churn in defect-fixing commits (DFC) for both smelly and non-smelly IaC files for small, medium, and large files, as reported in Figure 8. An average code churn of 8.57 and an average of 9.91 is witnessed in small-sized smelly and non-smelly IaC files with a non-significant p-value of 0.13 and negligible effect size. The average code churn score for medium size files is 14.7 in smelly IaC files and 14.17 in non-smelly IaC files, with a non-significant p-value of 0.54 and negative negligible effect size. Large smelly IaC files undergo an average of 25.43 of code churn compared to an average of 49.51 in non-smelly IaC files, with non-significant p-value and negative negligible effect size. Regardless of their size, smelly IaC files witness lower code churn than non-smelly IaC files. These results foster the understanding of types of defects that are likely to manifest in smelly IaC files compared to non-smelly IaC files.

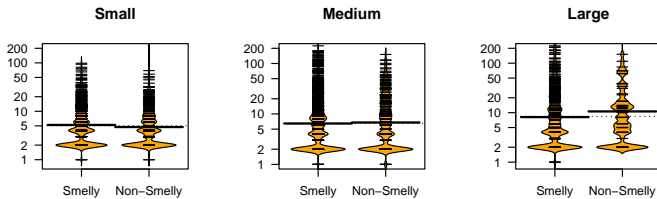


Fig. 8: Distribution of code changes (churn) in defect-fixing commits in smelly and non-smelly files for small, medium, and large sized files.

## V. THREATS TO VALIDITY

**Internal threats to validity** may affect the integrity of our study. In RQ2, we do not make the explicit assumption that when IaC smells frequently appear together, one smell causes the other. Consequently, we cannot definitively conclude that a specific smell is the root cause of another smell being present. Besides, our findings concerning how IaC smells influence changes and defects do not necessarily imply a direct cause-and-effect relationship. Other factors, such as refactoring or general code improvements, can also impact these outcomes. **Construct threats to validity** can impact the alignment between the theoretical constructs and measures. To collect our dataset, we relied on the accuracy of the Puppeteer tool [16]. However, there could still be errors in the detection of IaC smells. Another potential threat to validity could be related to the identification of defect-fixing commits using keywords. Although this technique has been widely used in previous studies [39, 40], it could not be free of false positives when finding defect-inducing commits. To mitigate this issue, we randomly selected and analyzed a representative sample of 375 defect-fixing commits and their related defect-inducing commits and found a precision over 85% and 97%, respectively.

**External threats to validity** could impact the generalizability of our study findings. To the best of our knowledge, this is the first empirical analysis conducted on the prevalence, co-occurrence, and impact of IaC smells in Puppet projects. While we considered 82 projects and 12 IaC smells, we cannot generalize our results to other projects.

## VI. CONCLUSION

In this research paper, we presented an empirical investigation involving 82 Puppet projects to examine the prevalence and co-occurrence patterns of 12 types of IaC smells, and to assess how these IaC smells impact the files’ proneness to changes and defects. Our findings reveal that IaC smells are notably prevalent and frequently co-occur within Puppet projects. For instance, certain IaC smells, such as “*Insufficient Modularization (DIM)*” are pervasive. Furthermore, our analysis indicates that nearly half of the smelly IaC files exhibit multiple instances of IaC smells. For example, the “*Deficient encapsulation (DDE)*” is often associated with three distinct types of smells, including “*Unstructured module (DUM)*” and “*Dense Structure (DDS)*”. Furthermore, we found that smelly IaC files are 3.8 times more prone to be modified, with a 3.1 times higher susceptibility to code changes compared to non-smelly files. As for the defect-proneness, smelly files are 3.3 times more prone to defects that are likely to persist in 1.65 more commits before being fixed than non-smelly files. Nevertheless, the churn required to fix these defects remains relatively the same for both smelly and non-smelly files. Our insights solicit developers to proactively detect and rectify IaC smells and foster IaC’s best coding practices, particularly in a rapidly evolving technology landscape. As part of our future directions, we plan to delve into the specific types of IaC smells impact on the change proneness and bug proneness.

<sup>9</sup><https://github.com/voxpupuli/puppet-zabbix/blob/master/manifests/agent.pp>

<sup>10</sup><https://github.com/voxpupuli/puppet-zabbix>

Furthermore, we plan to develop an automated approach to fix IaC smells.

## REFERENCES

- [1] Redhat, “What is infrastructure as code (iac)?” <https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac>, 2022.
- [2] J. Turnbull and J. McCune, *Pro Puppet*, vol. 1. Springer, 2011.
- [3] L. Hochstein and R. Moser, *Ansible: Up and Running: Automating configuration management and deployment the easy way*. O’Reilly Media, Inc., 2017.
- [4] Y. Brikman, *Terraform: Up & Running: Writing Infrastructure as Code*. O’Reilly Media, 2019.
- [5] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, “Adoption, support, and challenges of infrastructure-as-code: Insights from industry,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 580–589, IEEE, 2019.
- [6] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, “A systematic mapping study of infrastructure as code research,” *Information and Software Technology*, vol. 108, pp. 65–77, 2019.
- [7] M. Begoug, N. Bessghaier, A. Ouni, E. Alomar, and M. W. Mkaouer, “What do infrastructure-as-code practitioners discuss: An empirical study on stack overflow,” in *Proceedings of the 17th International Conference on Empirical Software Engineering and Measurement, ESEM ’23*, 2023.
- [8] M. Fowler and K. Beck, “Refactoring: Improving the design of existing code,” in *11th European Conference. Jyväskylä, Finland*, 1997.
- [9] K. Moris, “Infrastructure as code dynamic systems for the cloud age,” 2021.
- [10] R. Shambaugh, A. Weiss, and A. Guha, “Rehearsal: A configuration verification tool for puppet,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 416–430, 2016.
- [11] I. Kumara, M. Garriga, A. U. Romeu, D. Di Nucci, D. A. Tamburri, W.-J. van den Heuvel, and F. Palomba, “The do’s and don’ts of infrastructure code: A systematic grey literature review,” *Information and Software Technology*, p. 106593, 2021.
- [12] A. Rahman, C. Parnin, and L. Williams, “The seven sins: Security smells in infrastructure as code scripts,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 164–175, IEEE, 2019.
- [13] 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)*, pp. 220–228, IEEE, 2018.
- [14] 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 international conference on software analysis, evolution and reengineering (SANER)*, pp. 164–174, IEEE, 2018.
- [15] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, “Towards a catalogue of software quality metrics for infrastructure code,” *Journal of Systems and Software*, p. 110726, 2020.
- [16] T. Sharma, M. Fragkoulis, and D. Spinellis, “Does your configuration code smell?,” in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 189–200, IEEE, 2016.
- [17] A. Rahman and T. Sharma, “Lessons from research to practice on writing better quality puppet scripts,” in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 63–67, IEEE, 2022.
- [18] A. Rahman and C. Parnin, “Detecting and characterizing propagation of security weaknesses in puppet-based infrastructure management,” *IEEE Transactions on Software Engineering*, 2023.
- [19] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, “On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation,” *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.
- [20] N. Bessghaier, A. Ouni, and M. W. Mkaouer, “A longitudinal exploratory study on code smells in server side web applications,” *Software Quality Journal*, vol. 29, pp. 901–941, 2021.
- [21] A. Chatzigeorgiou and A. Manakos, “Investigating the evolution of bad smells in object-oriented code,” in *Seventh International Conference on the Quality of Information and Communications Technology*, pp. 106–115, 2010.
- [22] F. Palomba, R. Oliveto, and A. De Lucia, “Investigating code smell co-occurrences using association rule learning: A replicated study,” in *IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pp. 8–13, 2017.
- [23] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, “A large-scale empirical study on the lifecycle of code smell co-occurrences,” *Information and Software Technology*, vol. 99, pp. 1–10, 2018.
- [24] B. Asmare Muse, M. Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, “On the prevalence, impact, and evolution of sql code smells in data-intensive systems,” in *International Conference on Mining Software Repositories (MSR 2020)*, 2020.
- [25] O. Hamdi, A. Ouni, E. A. AlOmar, and M. W. Mkaouer, “An empirical study on code smells co-occurrences in android applications,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pp. 26–33, IEEE, 2021.
- [26] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, “Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source

- systems,” in *IEEE International Conference on Software Maintenance*, pp. 1–10, 2010.
- [27] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, “An exploratory study of the impact of code smells on software change-proneness,” in *Working Conference on Reverse Engineering*, pp. 75–84, 2009.
- [28] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, “On the relation of test smells to software code quality,” in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 1–12, 2018.
- [29] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol, “An empirical study of code smells in javascript projects,” in *International conference on software analysis, evolution and reengineering (SANER)*, pp. 294–305, 2017.
- [30] “Replication package for the paper: ”on the prevalence, co-occurrence, and impact of infrastructure-as-code smells”.” <https://zenodo.org/records/10060209>. Accessed on: May 31, 2023.
- [31] R. Agrawal, R. Srikant, *et al.*, “Fast algorithms for mining association rules,” in *20th international conference on very large data bases, VLDB*, vol. 1215, pp. 487–499, 1994.
- [32] R. Agrawal, T. Imieliński, and A. Swami, “Mining association rules between sets of items in large databases,” in *ACM SIGMOD international conference on Management of data*, pp. 207–216, 1993.
- [33] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, “Dynamic itemset counting and implication rules for market basket data,” in *ACM SIGMOD international conference on Management of data*, pp. 255–264, 1997.
- [34] M. L. McHugh, “The chi-square test of independence,” *Biochemia medica: Biochemia medica*, vol. 23, no. 2, pp. 143–149, 2013.
- [35] H. Cramér, “Mathematical methods of statistics,” *Princeton U. Press, Princeton*, p. 500, 1946.
- [36] D. Spadini, M. Aniche, and A. Bacchelli, “Pydriller: Python framework for mining software repositories,” in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 908–911, 2018.
- [37] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su, “Exploring and exploiting the correlations between bug-inducing and bug-fixing commits,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 326–337, 2019.
- [38] J. Sliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?,” *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.
- [39] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. Tamburri, “Within-project defect prediction of infrastructure-as-code using product and process metrics,” *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 01 2021.
- [40] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, “Towards building a universal defect prediction model,” in *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014*, (New York, NY, USA), p. 182–191, Association for Computing Machinery, 2014.
- [41] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, “A framework for evaluating the results of the szz approach for identifying bug-introducing changes,” *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.
- [42] Y. Fan, X. Xia, D. A. da Costa, D. Lo, A. E. Hassan, and S. Li, “The impact of mislabeled changes by szz on just-in-time defect prediction,” *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1559–1586, 2021.
- [43] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead, “Automatic identification of bug-introducing changes,” in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*, pp. 81–90, 2006.
- [44] E. C. Neto, D. A. da Costa, and U. Kulesza, “The impact of refactoring changes on the szz algorithm: An empirical study,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 380–390, 2018.
- [45] W. J. Conover, *Practical nonparametric statistics*, vol. 350. John Wiley & Sons, 1998.
- [46] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, “Cliff’s delta calculator: A non-parametric effect size program for two groups of observations,” *Universitas Psychologica*, vol. 10, no. 2, pp. 545–555, 2011.
- [47] M. G. Kendall, “A new measure of rank correlation,” *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.
- [48] N. Bessghaier, A. Ouni, and M. W. Mkaouer, “On the diffusion and impact of code smells in web applications,” in *Services Computing–SCC 2020: 17th International Conference, Held as Part of the Services Conference Federation, SCF 2020, Honolulu, HI, USA, September 18–20, 2020, Proceedings 17*, pp. 67–84, Springer, 2020.
- [49] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, “An exploratory study of the impact of antipatterns on class change-and fault-proneness,” *Empirical Software Engineering*, pp. 243–275, 2012.