When Your Infrastructure Is a Buggy Program: Understanding Faults in Infrastructure as Code Ecosystems

GEORGIOS-PETROS DROSOS*, ETH Zurich, Switzerland
THODORIS SOTIROPOULOS*, ETH Zurich, Switzerland
GEORGIOS ALEXOPOULOS, University of Athens, Greece
DIMITRIS MITROPOULOS, University of Athens, Greece
ZHENDONG SU, ETH Zurich, Switzerland

Modern applications have become increasingly complex and their manual installation and configuration is no longer practical. Instead, IT organizations heavily rely on *Infrastructure as Code (IaC)* technologies, to automate the provisioning, configuration, and maintenance of computing infrastructures and systems. IaC systems typically offer declarative, domain-specific languages (DSLs) that allow system administrators and developers to write high-level programs that specify the desired state of their infrastructure in a reliable, predictable, and documented fashion. Just like traditional programs, IaC software is not immune to faults, with issues ranging from deployment failures to critical misconfigurations that often impact production systems used by millions of end users. Surprisingly, despite its crucial role in global infrastructure management, the tooling and techniques for ensuring IaC reliability still have room for improvement.

In this work, we conduct a comprehensive analysis of 360 bugs identified in IaC software within prominent IaC ecosystems including Ansible, Puppet, and Chef. Our work is the first in-depth exploration of bug characteristics in these widely-used IaC environments. Through our analysis we aim to understand: (1) how these bugs manifest, (2) their underlying root causes, (3) their reproduction requirements in terms of system state (e.g., operating system versions) or input characteristics, and (4) how these bugs are fixed. Based on our findings, we evaluate the state-of-the-art techniques for IaC reliability, identify their limitations, and provide a set of recommendations for future research. We believe that our study helps researchers to (1) better understand the complexity and peculiarities of IaC software, and (2) develop advanced tooling for more reliable and robust system configurations.

CCS Concepts: • **Computer systems organization** → **Reliability**; • **Software and its engineering** → **Software testing and debugging**; **System administration**.

Additional Key Words and Phrases: IaC, infrastructure as code, bug, Puppet, Ansible, Chef, testing, deployment

ACM Reference Format:
Georgios-Petros Drosos, Thodoris Sotiropoulos, Georgios Alexopoulos, Dimitris Mitropoulos, and Zhendong Su. 2024. When Your Infrastructure Is a Buggy Program: Understanding Faults in Infrastructure as Code Ecosystems. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 359 (October 2024), 31 pages. https://doi.org/10.1145/3689799

*These authors contributed equally.

Authors' Contact Information: Georgios-Petros Drosos, ETH Zurich, Zurich, Switzerland, gdrosos@student.ethz.ch; Thodoris Sotiropoulos, ETH Zurich, Zurich, Switzerland, theodoros.sotiropoulos@inf.ethz.ch; Georgios Alexopoulos, University of Athens, Athens, Greece, grgalex@ba.uoa.gr; Dimitris Mitropoulos, University of Athens, Athens, Greece, dimitro@ba.uoa.gr; Zhendong Su, ETH Zurich, Zurich, Switzerland, zhendong.su@inf.ethz.ch.

ACM 2475-1421/2024/10-ART359
https://doi.org/10.1145/3689799

1 Introduction

Modern applications are no longer monolithic pieces of code that are tested and installed in an isolated, static environment, such as a bare-metal server or a virtual machine. Instead, they consist of many highly interconnected components including software dependencies, database management systems and load-balancing solutions. In turn, these components are deployed in complex and dynamic environments that are based on large computing and network infrastructures. The provision of such IT environments and the deployment of modern applications within them has become increasingly complex [Delaet et al. 2010; Morris 2016; Spinellis 2012].

The *Infrastructure as Code (IaC)* paradigm has introduced a number of technologies to address the aforementioned challenges. IaC is a term that describes IT solutions that automate the process of provisioning, configuring, and managing computing infrastructures by using declarative configuration files. In this context, IaC systems provide high-level *domain-specific languages (DSLs)* that allow users to write programs that specify the desired state of their infrastructure. The automation provided by IaC significantly minimizes human intervention. This reduces the likelihood of human errors and ensures consistent and reliable deployments [Morris 2016; Spinellis 2012; Visser et al. 2016]. IaC systems are currently widely adopted by organizations [Artac et al. 2017; Guerriero et al. 2019], and supported by numerous thriving ecosystems [Ansible 2024a]. These ecosystems allow developers and system administrators to leverage existing IaC software for their own infrastructure and deployments.

Given that the software deployed in IaC ecosystems manages global infrastructures, its reliability is of paramount importance. However, just like traditional software, IaC software is not free of bugs. Such bugs can lead to critical issues that range from frustrating deployment failures to dangerous system and infrastructure misconfigurations. Furthermore, IaC software bugs frequently impact deployed systems that are used in production by millions of end users. A buggy IaC software can lead to devastating and costly production incidents, such as outages [Amazon Web Services, Inc. or its affiliates 2017; GitHub, Inc. 2014], data losses [Wikimedia Commons 2017], or security issues [Lepiller et al. 2021; Rahman et al. 2019].

Despite its critical role, it is surprising that there is limited tooling available to improve the reliability of IaC software. Currently, industry practices primarily rely on traditional testing methods, such as unit testing or integration testing, if testing is conducted at all [Pulumi 2024]. Yet, these testing efforts can fail to detect subtle IaC bugs, mainly because the manually-written tests often omit edge cases.

In this study, we present the first quantitative and qualitative analysis of bugs in software deployed in IaC ecosystems. Our objective is to explore and understand how these bugs manifest, and distill knowledge about their root causes, triggers, and fixes. Our work aims to answer the following research questions:

RQ1 (Symptoms) What are the main symptoms of IaC bugs? What is the frequency of these symptoms? (Section 4.1)

RQ2 (Bug Causes) What are the main causes of IaC bugs? Can we identify common patterns and group causes into categories? What is the frequency of these categories? (Section 4.2)

RQ3 (System State and Test Inputs) How are IaC bugs triggered? Is it easy to trigger these bugs? Do they depend on the state of the underlying system (e.g., specific operating system versions)? What kind of test input is required to trigger these bugs? (Section 4.3)

RQ4 (Bug Fixes) How are IaC bugs fixed? What are the components affected by the fixes? (Section 4.4)

To answer these questions, we study bugs found in software written to run in popular IaC systems. Currently, there are a plethora of IaC systems including Chef [Progress 2024], Puppet [Perforce

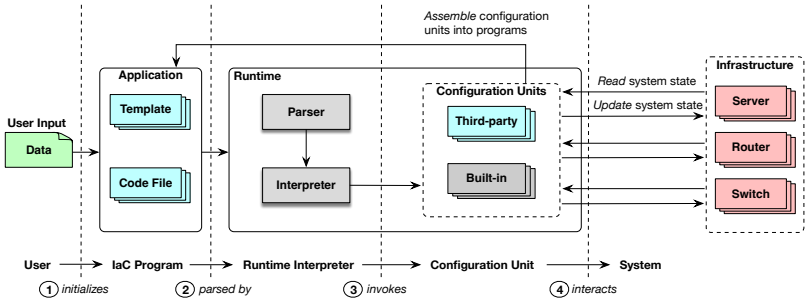


Fig. 1. The architecture of configuration-based IaC tools.

2024], Ansible [RedHat, Inc. 2024], or Terraform [HashiCorp 2024]. Each of these systems may have different goals and purposes. For example, Terraform is mainly used for provisioning and managing cloud resources such as computing instances within a cloud (e.g., AWS), whereas Puppet, Ansible, and Chef are mostly known for configuration management of resources (e.g., files, network) on individual systems that are parts of a broader network and/or computing infrastructure. In this work, we consider only configuration-based IaC systems, namely, Ansible, Puppet, and Chef. Configuration-based IaC systems hold a significant stake in the IaC technology and they have been subjects of extensive research over the past ten years [Guerriero et al. 2019; Opdebeeck et al. 2023a; Rahman et al. 2020, 2019; Saavedra and Ferreira 2023; Shambaugh et al. 2016; Sharma et al. 2016; Sotiropoulos et al. 2020; Weiss et al. 2017]. From now on, and unless specified otherwise, we use the term IaC to refer to configuration-based IaC systems.

Focusing on the ecosystems of Ansible, Puppet, and Chef, we use carefully-crafted criteria to search for *resolved* bug reports in IaC software deployed in these ecosystems. Our bug collection method results in a dataset consisting of 24,807 IaC bugs along with their fixes. We manually analyze a random sample of 360 IaC bugs and evaluate each one in terms of its (1) symptoms, (2) causes, (3) system and input requirements, and (4) fixes.

Contributions: Our work makes the following contributions:

- We present the first systematic study of IaC bugs taken from popular ecosystems, and create a corresponding reference dataset containing 24,807 issues (Section 3).
- We introduce a thorough taxonomy from the analysis of 360 IaC bugs. Our taxonomy considers several aspects of IaC bugs, such as, their manifestations, root causes, reproduction requirements, and fixes (Section 4).
- We discuss the state-of-the-art techniques in IaC reliability, identify their gaps and limitations, and propose a set of guidelines for future research on enhancing IaC reliability (Section 5).

Summary of findings: Some key findings from our study include: (1) IaC bugs mainly lead to abrupt program terminations (37%), often due to external command failures or network issues during remote communications; (2) Nearly half (45%) of IaC bugs lie in logics related to system interaction and system state manipulation; (3) More than half (52%) of the bugs depend on the target system's initial state, with 37% of these bug-enabling states being recreated *entirely* through specific inputs to the IaC code under test; and (4) IaC bugs are typically fixed via small code modifications that involve a few lines of code (median of eight) in a single source file.

2 Background

This section provides a brief overview of the IaC system architecture and the state-of-the-art IaC tools, and concludes by summarizing the scope of this study.

```
package {"Install apache2 package":
  name => "apache2"
  ensure => "installed"
}

file {"Configure apache2 conf":
  ensure => "file"
  path => $conf_file
  contents => template("files/apache.erb")
}

service {"Start apache2 service":
  name => "apache2"
  ensure => "running"
}
```

(a) An example program written in Puppet.

```
- name "Install apache2 package"
  ansible.builtin.package:
    name: "apache2"
    state: installed

- name "Configure apache2 conf"
  ansible.builtin.template:
    src: "files/apache.j2"
    dest: "{{ conf_file }}"

- name "Start apache2 service"
  ansible.builtin.service:
    name: "apache2"
    state: started
```

(b) An example program written in Ansible.

Fig. 2. Install and configure an Apache server using Puppet and Ansible.

2.1 Architecture of IaC Applications

Figure 1 illustrates the high-level architecture of IaC, which is divided into two primary components: *application layer* and *runtime*. At the application layer, developers write *high-level programs* using DSLs offered by IaC systems. These programs are essentially the specification that describes the desired state of the system. The IaC system parses, interprets, and finally executes these programs. Then, it engages in numerous interactions with the target system to align its actual state with the specified desired state.

Runtime: The fundamental entity of the *runtime* component is the *configuration unit*. A configuration unit is responsible for interacting with the target system. We define a target system as any receiver of configuration unit actions, be it a Linux host, a network device, or even a Cloud platform API. Conceptually, a configuration unit models the state of a component in the system. Such components include files, network interfaces, or running processes. Configuration units are programs written in conventional programming languages (e.g., Python) that expose an API that is used to specify the intended state of these system components. Then, configuration units operate in a three-step process: (1) they read part of the current system state, (2) identify deviations between the current and the desired state, and finally (3) apply necessary modifications to align the system with the desired state. The runtime comes with a collection of built-in configuration units for managing files, packages, or services. However, it is possible to extend the runtime component with third-party configuration units which typically abstract arbitrary aspects of the system. Such aspects include the state of a database, or a Docker container.

Application layer: To invoke a configuration unit, one needs to write an *IaC program* using the DSL provided by the IaC system. IaC programs are essentially a structured combination of various configuration unit invocations. Each invocation specifies a particular action or desired state for different parts of the system's infrastructure. Executing an IaC program triggers a sequence of these unit invocations that configure, manage, or alter system resources, according to the logic defined in the program. To facilitate advanced orchestration of configuration units, IaC DSLs incorporate features that appear in traditional programming languages, such as variables, conditionals, loops, classes, and more. Finally, beyond DSL code, an IaC program might contain templates of configuration files. Templates represent text files whose contents can be modified based on the data passed by the user. Templates allow the creation of complex files (e.g., a configuration file for a web server) dynamically.

Example: Figure 2 presents two IaC programs written in the DSLs of Puppet and Ansible. These programs illustrate the process of installing the `apache2` package, creating its configuration file, and spawning the corresponding service. To do so, these programs invoke three *configuration units* to

manage the respective system resources. For OS package management, the code uses the `package` unit of Puppet (Figure 2a, lines 1–4) and the `ansible.builtin.package` unit of Ansible (Figure 2b, lines 1–4) respectively. To create the Apache configuration file the code invokes Puppet's `file` unit, with the file's location and contents determined by a `conf_file` variable and a template (Figure 2a, lines 6–10). In a similar way, the Ansible program uses the `ansible.builtin.template` configuration unit to achieve a similar functionality (Figure 2b, lines 6–9). Both programs employ a `service` configuration unit to ensure that an Apache service is operational. Notice that calling a configuration unit requires supplying a set of arguments that adhere to its API. For example, the Puppet's `service` configuration unit expects a parameter called `ensure` (Figure 2a, line 14), which specifies the status of the `apache2` service (active status). Finally, program variables and templates are populated with user-provided data during execution.

Relation of IaC with cloud management software: The high-level architecture of IaC shares similarities with container orchestration systems such as Kubernetes. In Kubernetes [The Kubernetes authors 2024], operators and controllers [Gu et al. 2023; Sun et al. 2022] automate the management of complex stateful applications deployed on a Kubernetes cluster. These controllers and operators continuously monitor the cluster's state and make changes to align it with the desired state defined by declarative configuration files. A key difference between IaC and Kubernetes controllers/operators lies in their scope and focus. Kubernetes controllers and operators manage the lifecycle of resources within the Kubernetes ecosystem, *exclusively* using the core Kubernetes API. Through this API, they handle tasks such as application scaling, update roll-outs, and failure recovery. Nevertheless, their operations are confined to the Kubernetes environment. In contrast, IaC offers a broader approach that manages a broader infrastructure landscape that includes databases, network devices, cloud hosts, and bare-metal servers.

2.2 IaC Tools

Among the many configuration-based IaC solutions, Ansible, Puppet, and Chef stand out as the most popular according to Stackoverflow's annual developer survey [Stack Exchange Inc. 2023]. Puppet first appeared in 2005. It is implemented in Ruby and provides its own declarative DSL with features such as classes, inheritance, conditionals and loops. Chef is built in Ruby and Erlang, while it offers a Ruby-like DSL. Regarding Ansible, although it is a relatively new IaC system (it first appeared in 2012), it has gained much popularity recently. Ansible has been developed in Python, and its DSL relies on the Yet Another Markup Language (YAML) format.

Table 1. List of IaC terms as they are defined in every system.

Component	Puppet	Ansible	Chef
Configuration unit	Resource	Module	Resource
IaC program	Module	Role	Cookbook

All these systems support vast and thriving ecosystems that enable the reuse of existing IaC programs and configuration units. Notably, the ecosystem of Ansible includes more than 30k configuration units and Ansible programs that manage the state of miscellaneous resources of cloud or network infrastructures. While IaC programs are implemented using the specific DSLs provided by these systems (e.g., Puppet DSL), the underlying configuration units are written in conventional programming languages, such as Ruby, Python, or PowerShell. Table 1 correlates each term with its counterpart in the studied systems. For example, in Ansible, "modules" and "roles" are configuration units and Ansible programs, respectively. In the remainder of the paper, we may use IaC-specific terminology when discussing bugs.

Scope: In this work, we delve into the bugs that are prevalent within IaC ecosystems. We are interested in bugs that affect both the *application layer* and *runtime*. Within the application layer, we focus on bugs in Puppet modules, Ansible roles, and Chef cookbooks. Regarding the runtime, we

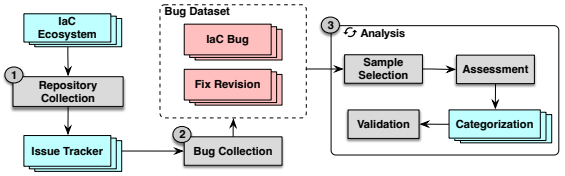


Fig. 3. Our methodology for collecting and analyzing bugs.

Table 2. Statistics on our bug collection and analysis. Each table entry provides descriptive statistics for each ecosystem. For the bug collection phase it presents: (1) the total number of repositories (Total Repositories) after the *repository collection* (RC) step, (2) the number of potential bugs (Total Issues) identified after executing the *bug collection* step (BG), and (3) the creation date of the oldest and most recent issue considered in the study (Oldest/Most Recent). For the bug analysis phase (360 bugs), it shows the distribution of the selected bugs split by IaC component, i.e., configuration units vs. IaC programs.

Ecosystem	Total Repositories	Total Issues	Oldest	Most Recent	Config. Unit Bugs	IaC Program Bugs
Puppet	7,471	6,750	6 Aug 2013	2 Feb 2023	42	78
Ansible	35,236	16,916	19 Sep 2014	3 Oct 2023	94	26
Chef	2,818	1,141	23 Aug 2011	9 March 2023	76	44

are interested in configuration unit bugs, which concern the implementation of Puppet resources, Ansible modules, and Chef resources.

3 Methodology

First, we present our approach for creating a dataset of bugs found in the ecosystems of Puppet, Ansible, and Chef (Section 3.1). Then, we explain how we study and analyze our dataset (Section 3.2), and finally we discuss threats to validity of our bug collection and analysis approach (Section 3.3).
Overview: Figure 3 outlines our methodology for collecting and analyzing IaC bugs. Initially, we take the official package registry of every IaC system studied in this work, and we examine the metadata of every artifact hosted in these registries. Focusing on this metadata, we are able to identify the issue trackers linked to each IaC artifact. In this way, we compile a collection that includes the issue trackers of all the code deployed within IaC ecosystems (*repository collection*). Then, during the *bug collection* phase, we query every issue tracker taken from the first step, and search for issues that are (1) fixed, and (2) linked to a pull request or a commit. The resulting dataset consists of bug reports and their associated fix revisions (Section 3.1).

The next step involves the analysis of our bug dataset (Section 3.2). The analysis is done in an iterative fashion. In each iteration, a random sample of n bugs is picked (*sample selection*). Then, two co-authors independently study every bug in the sample and evaluate it to answer RQ1–RQ4 (*analysis*). The outcome of the analysis is a characterization of the examined bugs, which is further validated by an additional co-author (*validation*).

3.1 Collecting Bugs and Fixes

Our bug collection method consists of two phases: *repository collection* and *bug collection*. In the first phase, we systematically look for *both* reusable IaC programs (*application layer*) and configuration units (*runtime*) in the official package registries of the IaC systems. Using the REST APIs of the package registries [Ansible 2024a; Chef Software, Inc. 2024b; Puppet 2024a], we fetch metadata for all the deployed IaC entities, particularly their issue trackers. This search yields three separate sets: R_p, R_a, and R_c. These sets contain the URLs of the issue trackers linked to programs within the Puppet, Ansible, and Chef ecosystems, respectively.

Having retrieved the issue trackers of the artifacts within the IaC ecosystems, we proceed with the *bug collection* phase. Our goal is to apply a series of filtering criteria to extract bug-related

issues from the issue trackers gathered from our initial phase (*repository collection*). In particular, our search criteria include issues that (1) are closed, and (2) are linked to at least one pull request or commit, indicating an available fix. To do so, we employ the GitHub GraphQL API, which provides advanced query capabilities that are suitable for processing massive datasets. Note that the development teams of the examined IaC programs employ different labeling mechanisms for distinguishing bugs from other issues (e.g., feature requests), or they do not employ any labeling at all. For these reasons, our aforementioned search criteria remain general. For example, we avoid fetching only the issues marked with the label "bug". Because of the generality of our search criteria, our final dataset might contain issues that are not bugs. However, during our bug analysis approach and after careful examination (Section 3.2), we analyze only issues that are deemed to be bugs.

Table 2 provides a summary of statistics regarding our dataset. Specifically, our final dataset includes 6,750 issues from Puppet modules and resources, 16,916 bugs from Ansible modules and roles, and 1,141 bugs from Chef resources and cookbooks. These bugs were identified from the analysis of over 45k IaC artifacts, of which 7,471 are Puppet artifacts, 35,236 are Ansible artifacts, and 2,818 are Chef artifacts.

Remark on Puppet bug collection: One challenge that we faced during the collection of Puppet bugs is that some modules maintained by the Puppet development team were hosted in a Jira issue tracker, but later transitioned to GitHub. As a result, some artifacts hosted in Forge, the official package registry of Puppet, contained both a GitHub and a Jira URL. To address this, we wrote two queries to fetch relevant bugs: one for the GitHub issue tracker using GraphQL, and one for Jira using the Jira Query Language (JQL). In total, we successfully collected 6,750 Puppet bugs (Table 2), of which, 2,143 bugs came from a Jira issue tracker.

3.2 Analyzing Bugs

Our bug analysis mostly relies on the manual examination of the collected bug reports and fixes. A manual analysis is typically costly and requires considerable time and human effort. To address this challenge, we answer the research questions based on a sample of 360 bugs chosen at random from our original bug dataset. To ensure our findings are generalizable and avoid bias towards any specific IaC system, our sample equally represents each IaC system, i.e., 120 bugs from each system.

Manual and qualitative bug analysis: To answer RQ1–RQ3, we manually analyzed our bugs in an iterative fashion. In each iteration, we randomly selected 20 bugs from the bug set of every IaC system. Two co-authors independently examined every selected bug and tried to assign them into categories according to the study's research questions. To answer RQ1, the two co-authors inspected the description of every bug report, and assessed the differences between the expected and the actual behavior of the buggy IaC code. Regarding RQ2, the authors considered both the bug reports and the accompanying fixes to identify common patterns of problematic procedures in IaC code. Finally, for RQ3, the authors reviewed (1) the reproduction steps found in the body of the selected bug reports, and (2) the associated bug fixes. Based on these, the authors evaluated each bug in terms of (1) OS requirements, (2) system state requirements (e.g., a specific file must exist in the system before running the IaC program), and (3) characteristics of inputs.

Assessing OS and state requirements can be challenging. The authors examined the official documentation of IaC packages to extract the list of supported OSs. For example, the Forge API maintains metadata about the deployed Puppet modules. By inspecting the `current_release.metadata.operatingsystem_support` field in the metadata of a specific Puppet module, one can find its supported OSs. To determine whether a specific OS is required for triggering a bug, in addition to the informative bug reports (reproduction steps), the authors examined the corresponding bug fix. In most cases, finding this information was straightforward due to the modular nature of the studied programs, which store OS-specific code in dedicated files.

Note that there was a small number of cases, where the authors scrutinized the fixing commits to deduce if they implemented OS- or state-specific changes.

After their independent bug examination, the two co-authors together discussed the categorization until they reached consensus. The entire process was repeated six times until having analyzed 360 in total. In the beginning, there was no predefined category. After thorough discussions, the authors created new categories or renamed, split, merged and adapted existing ones as needed. Notably, the difficulty of categorizing bugs gradually decreased after the iterative refinement of our categories. Finally, to mitigate the threat of misclassification and further validate the process, an additional co-author classified the bugs based on the proposed categorization.

Automated and quantitative bug analysis: Our approach for RQ4 was fully automated. For each bug, we retrieved the GitHub commit or pull request linked to its fix, and used the "commits/" and "pull/" endpoints of the GitHub API to measure the lines of code and the files affected by the fix. We focused only on the source files of the IaC programs. Revisions related to test code (e.g., unit tests) or documentation files were intentionally excluded from the analysis.

To distinguish between fixed files in the two core components of IaC systems (application layer vs. runtime), we introduced a file classification method in our analysis, which adheres to a predefined directory structure thoroughly detailed in the documentation of each ecosystem [Ansible 2024b; Chef Software, Inc. 2024a; Puppet 2024b]. Based on the documentation, we identified the directories which host IaC programs or configuration units. Given this information, we automatically classified each file based on its directory path. For example, files existing within the "roles/" directory in Ansible were classified as the source code of Ansible programs.

Based on this classification, Table 2 shows that 59% (212/360) bugs were found in configuration units, while the remaining 41% of the analyzed bugs appear in IaC programs. Notably, Ansible and Chef show a higher frequency of bugs in configuration units (94/120 and 76/120, respectively), whereas the majority of Puppet bugs lie in IaC programs (78/120).

3.3 Threats to Validity

Selection criteria: The selection criteria we employed for fetching IaC bugs could be a potential threat. Our analysis focuses on studying fixed bugs in IaC code, which we identified by selecting closed issues linked with a pull request from IaC package repositories. Not all issues that satisfy the aforementioned criteria are necessarily a bug; some might be features or other non-bug-related changes. To ensure that the focus remains on actual bugs, during our manual analysis, we carefully examined each selected issue. If the chosen issue was not a bug, we replaced it with another one from our bug dataset that met our criteria. Investigating real-world, fixed bugs aligns with the scope of other empirical studies [Bagherzadeh et al. 2020; Chaliasos et al. 2021; Eghbali and Pradel 2021; Xiong et al. 2023]. Fixed bugs are a valuable source of information for understanding their nature and their resolution.

Quality of bug reports: The quality of the studied bug reports can significantly impact our evaluation process, posing a potential threat to validity. Fortunately, most IaC development teams adhere to specific bug report guidelines, which ensure structured and detailed information, including the expected IaC program behavior, reproduction steps, OS and environment requirements (e.g., consider the bug reports of wwilldurand-puppet-nodejs-190 and ansible-collections/community.postgresql-314). During our random bug sample selection (Step 3, Figure 3), we excluded reports lacking informative descriptions. Such cases were replaced with more informative bugs that were also randomly chosen. To further mitigate this threat, beyond bug reports, we also examined the associated bug fixes, which is another rich source of information on root causes and OS requirements (Section 3.2).

Representativeness of selected bugs: The amount of bugs studied (360) could be a threat to the external validity, because the selected bugs might not accurately reflect the entire bug population. Given the time-intensive nature of our manual analysis, we selected a random sample aligned with our study's scope. Indeed, studying 360 bugs manually was challenging. Each bug required understanding its root causes, system state requirements, and fix, compounded by the diversity of IaC software functionalities as well as the programming languages (Puppet DSL, YAML, Ruby, Python) used in IaC ecosystems. As the first study of its kind, significant effort was needed to develop categories, especially for non-standard classifications such as system state requirements. Manually analyzing 360 bugs is consistent with state-of-the-art bug studies [Bagherzadeh et al. 2020; Chaliasos et al. 2021; Di Franco et al. 2017; Eghbali and Pradel 2021; Leesatapornwongsa et al. 2016; Xiong et al. 2023], which have analyzed a range between 100 to 400 bugs. In theory, when working with a random sample of 360 bugs, there is a 2.6% chance of missing a bug category whose relative frequency is at least 1%. Probabilities of this kind can be computed by the following formula (outlined in the work of Mastrangelo et al. [2019]): $(1 - \mathit{relative\_frequency})^{\mathit{sample\_size}} = (1 - 1\%)^{360} \approx 2.6\%$.

Another threat to external validity is the relevance of the studied bugs to the ones that plague IaC systems today. Our dataset includes bugs spanning more than a decade, from 2011 to 2023: 23 of them were fixed before 2015, 185 from 2015 to 2019, and the remaining 152 (42%) were resolved from 2020 onwards. To validate the relevance of our study, we split the dataset into bugs fixed before 2019 (179) and those fixed during and after 2019 (181). *Chi-Square tests* for each bug category showed no statistical differences between the two samples, indicating that the bugs studied before 2019 share the same characteristics as more recent bugs. Finally, adding more to the issue of the representativeness of the selected bugs, limiting our analysis to publicly disclosed bugs could raise another threat to validity. Certain types of bugs might never be made public, and thus, are not represented in the analysis.

Representativeness of the selected IaC software and IaC ecosystems: An additional threat to the external validity is the representativeness of the subject IaC code. To mitigate this threat, rather than focusing on bugs in a few specific IaC packages, we considered the bugs reported in *all* the IaC packages deployed in the official software repositories of Ansible [Ansible 2024a], Puppet [Puppet 2024a], and Chef [Chef Software, Inc. 2024b]. This translates to over 45k IaC packages with functionalities ranging from network configuration and file system configuration to the deployment of cloud services and Docker-related tasks.

A further threat to the external validity is the representativeness of the chosen IaC technologies and systems. We selected Ansible, Puppet, and Chef, because they play a significant role in the IaC market, and have been subjects of prior research [Guerriero et al. 2019; Opdebeeck et al. 2023a; Rahman et al. 2020, 2019; Saavedra and Ferreira 2023; Saavedra et al. 2023; Shambaugh et al. 2016; Sharma et al. 2016; Sotiropoulos et al. 2020; Weiss et al. 2017]. However, we argue that some of the findings of our study might not be generalizable to IaC systems that embrace a different philosophy. For example, there are provision-based IaC systems, such as Terraform [HashiCorp 2024] or Pulumi [Pulumi 2024], used for managing computing infrastructures within a cloud (e.g., AWS). Whether our findings hold on such IaC systems remains to be seen in future studies.

Manual bug examination: Our manual analysis on the selected bugs comes with the risk of misclassification due to bias. To mitigate this threat, two co-authors independently studied the bugs, and then later, they discussed their categorization until reaching agreement. Their categorization was further validated by an additional co-author. This extra step of validation follows the best practices in manual qualitative analyses [Chaliasos et al. 2021].

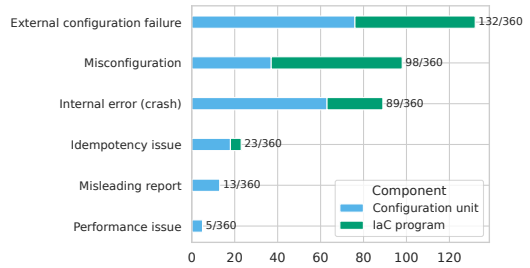


Fig. 4. Distribution of bug symptoms.

```
1       - name: Install Plugins
2       jenkins_plugin:
3         name: xvfb
4         url: "{{ jenkins_instance_url }}"
5         state: disabled
```

Fig. 5. Calling an Ansible module that triggers a bug with an *external configuration failure*.

4 Bug Study

We present our bug analysis results. We analyzed 360 IaC bugs, of which, 212 have been found in configuration unit implementations, and 148 in IaC programs written in high-level DSLs (Table 2). We allowed the data and analysis to guide our conclusions *without* any preconceived assumptions about the taxonomies of IaC programs and configuration units. Consequently, for some research questions, our findings revealed that IaC programs and configuration units share common categories (e.g., Section 4.1), while for others we identified unique categories specific to IaC programs, which are not applicable to configuration units (e.g., Section 4.2). Each answer to our research questions includes a *"Comparative Analysis"* section that discusses the differences and similarities between configuration units and IaC programs, supported by statistical tests.

4.1 RQ1: Symptoms

Every bug report of our dataset provides concise information regarding how a bug is triggered alongside the expected and actual behavior of the IaC system. We manually examined the differences between each program's expected and actual behavior and grouped them into distinct categories. This led us to identify six symptom categories, namely: *External Configuration Failure*, *Misconfiguration*, *Internal Error*, *Idempotency Issue*, *Misleading Report* and *Performance Issue*. Figure 4 presents the frequency of these categories. Notably, our findings show that IaC programs and configuration units share the same symptoms. This is because symptoms manifest at the end-user interface, where the effects of both IaC program bugs and configuration unit bugs are ultimately observed. As a result, despite differences in their underlying causes, the user experiences similar issues.

In the remainder of this section, we explore each category, discuss its prevalence and impact, and provide specific examples.

4.1.1 External Configuration Failure. *External configuration failure* is the most common symptom of our dataset (37%). It occurs when an IaC program fails to perform system-level operations or to communicate with remote services. The failures can be caused by various factors, including failures of external shell commands, network issues with remote communications, and more. When this symptom appears, users typically observe that their programs terminate gracefully, albeit unsuccessfully, providing error messages that indicate the nature of the external failure.

Example bug: ansible-collections/community.general-2510: Figure 5 presents an Ansible module (i.e., configuration unit) invocation that leads to an *external configuration failure*. Specifically, the code calls the Ansible module `jenkins_plugin`, which manages the state of Jenkins plugins. The intent of the code is to disable the `xvfb` plugin, as indicated by the `state` parameter (line 5). Upon execution, instead of the successful disablement of the plugin, the user receives an error message in the execution results: *"HTTP Error 405: Method Not Allowed"*. This issue arises because

the implementation of `jenkins_plugin` mistakenly sends an HTTP `GET` request, whereas the Jenkins API endpoint expects a `POST` request for state modifications, such as plugin disablements.

4.1.2 Misconfiguration. The second most common symptom of our dataset involves *misconfigurations* (27%). Such symptoms appear when an IaC program and the enclosing configuration units execute successfully (returning a zero exit code), yet fail to achieve the intended system state. Common misconfiguration outcomes include files created with incorrect contents/permissions, installation of wrong software packages, or services initialized with improper settings. Misconfigurations are challenging to diagnose because they do not result in immediate execution errors, but rather require thorough system inspection to verify the actual state against the desired configuration. From our symptom analysis, we observed that users detected the majority of misconfigurations when experiencing outages or malfunctions in the services impacted by these issues.
Example bug: voxpupuli/puppet-redis-425: Consider a Puppet module called voxpupuli/puppet-reddis whose main functionality involves the installation and management of a Redis database. A *misconfiguration* symptom appears in the following scenario: a user runs this program to deploy Redis using the default settings, while specifying a custom password for access control. Upon execution, the module terminates successfully, indicating no direct errors. Nevertheless, an oversight is discovered: the configuration file `/etc/redis/redis.conf`, which contains the password for the Redis database, is erroneously set to world-readable. This allows any local user, authorized or not, to access the database. In contrast, the expected behavior of the module is that the file is accessible exclusively by the Redis user and their group.

4.1.3 Internal Error (or Crash). Internal Error (or Crash) symptoms occur in 25% of the studied bugs. This symptom occurs when either an IaC program or a configuration unit terminates its execution abnormally. In contrast to an external configuration failure, which is triggered by unexpected issues when interacting with external systems, internal errors arise due to issues within the IaC code itself. Such errors lead to crashes and stacktraces, thereby failing to emit useful diagnostic messages or to bring the system to its desired state.
Example bug: voxpupuli/puppet-rabbitmq-704: Consider a Puppet program that manages RabbitMQ (a popular messaging and streaming broker) services. When a user invokes this Puppet program to install and configure RabbitMQ, they experience a Ruby-related crash with the following error message: *"Error: Facter: error while resolving custom fact "rabbitmq_version": undefined method '[]' for nil:NilClass"*. This internal error occurs when a Puppet resource (configuration unit) attempts to query the current version of RabbitMQ by running `rabbitmqadmin -version`. The error is caused by a bug in the official RabbitMQ's Debian package that returns a placeholder string (`%%VSN%%`) instead of a standard version number. This format is not anticipated by the configuration unit, and leads to its abrupt termination. The issue was resolved by modifying the Ruby code to ensure that it handles invalid version strings gracefully, preventing the crash.

4.1.4 Idempotency Issue. Idempotency [Couch and Sun 2003] is a fundamental property of IaC systems. This property dictates that applying a given configuration multiple times should leave the system in the same state as applying it just once, and without performing any redundant operations. This means that when a configuration is re-applied without any intervening modifications to the system, the program ought to recognize that no further changes are required to converge to the desired state. *Idempotency issues* constitute the 6% of the dataset and encompass bugs that exhibit the following property: executions, although successful, lead to different outcomes across repeated runs, or perform redundant operations. Examples include unnecessary service restarts, packages re-installed in each run, or files repeatedly created and deleted.

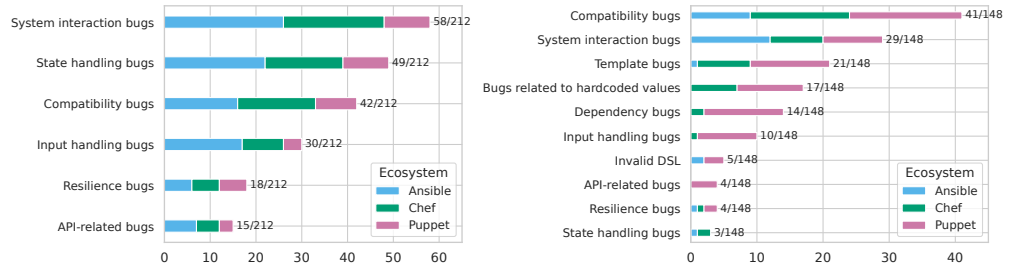
Example bug: sous-chefs/docker-1180: Consider a Chef resource named `docker_container` whose intention is to manage the state and the lifecycle of Docker containers. When calling this Chef resource to spawn a new container from a given image (e.g., `cadvisor:latest`), the execution creates the container as expected. However, when running `docker_container` for a second time, the container is unnecessarily restarted, violating the principle of idempotency. The resource mistakenly thinks that the container image has changed, which is caused by a bug in its logic for identifying discrepancies between the current state and the desired one.

Notably, in addition to configuration units, idempotency issues can be also found in IaC programs. This is because the DSLs of IaC technologies define specific constructs that control the invocation of the underlying configuration units. For example, Puppet's `unless` construct executes code only if a condition is false. Misusing these constructs can lead to idempotency issues. In puppetlabs/puppetlabs-docker-518, the Puppet module needlessly modified the Docker configuration file on every run, restarting the Docker daemon each time. This was caused due to an incorrect condition in a Puppet DSL `if` statement. The fix involved using the `unless` construct instead of `if` and refining the corresponding condition to prevent unnecessary modifications of the configuration file, thereby ensuring idempotency. In total, we encountered five idempotency issues in IaC programs.

4.1.5 Misleading Report. 4% of the bugs lead to a *misleading report* symptom. Such symptoms arise when the IaC execution outputs incorrect or misleading error messages, logs or warnings. It is important to note that in the case of a misleading report, the execution operates as intended, properly accepting or rejecting configurations, and executing the correct system operations. The issue lies solely in the generation of incorrect or misleading diagnostic messages.
Example bug: ansible-collections/community.general-561: This bug triggers a *misleading report* symptom, when calling `terraform`, an Ansible module, to provision ten instances on Google Cloud Platform (GCP). Under normal circumstances, Ansible should report: *"changed: [localhost]"* indicating that resources are indeed altered. However, a bug in the module's logic for interpreting Terraform's output led to a misinterpretation: it mistakenly reads *"10 added"* as *"0 added"* due to a pattern matching issue. Therefore, despite the successful resource deployment, Ansible erroneously reports: *"ok: [localhost]"* as if no changes occurred.

4.1.6 Performance Issue. Performance issues account for only 1% of the studied bugs. Such issues lead to significant performance degradation, characterized by increased memory consumption or extended execution times. Thus, programs run much longer than expected or, even worse, fail to terminate at all.
Example bug: ansible-collections/community.general-561: Consider an Ansible program that calls a buggy module (`win_powershell`) that triggers a *performance issue* symptom. The code invokes the `win_powershell` module to manipulate a log file on a Windows host using the execution of a PowerShell script. Here, the aim is to create a log file and return the file for further use in Ansible. However, calling this module causes a memory leak and the program hangs. The issue arises when the output of `Get-Content` (a PowerShell command), which is expected to be a simple string, is assigned to variable `$Ansible.Result`. However, each line returned by `Get-Content` is more than just a string, as it includes complex and recursive objects. The implementation of `win_powershell` triggers a memory leak when attempting to serialize these recursive objects. Fixing this bug required modifications in the system's serialization logic.

4.1.7 Comparative Analysis. Figure 4 indicates that *unexpected configuration failure*, *misconfiguration*, and *internal error* are the most frequent bug symptoms in both the application layer and runtime. Interestingly, *misconfigurations* are more frequently caused by IaC programs, in 62% of



(a) Distribution of root causes in configuration unit bugs.

(b) Distribution of root causes in IaC program bugs.

Fig. 6. Distribution of bug causes.

the cases, despite the fact that the majority of bugs in our dataset lie in the implementation of configuration units (Table 2). *Crashes* are more common in configuration units, accounting for 71% of such issues. This could be attributed to the fact that configuration units are written in conventional programming languages, which might be more susceptible to errors (e.g., `IndexError`). A *Chi-Square test* confirmed that configuration unit bugs and IaC program bugs are statistically different with respect to symptoms.

Regarding the distribution of symptoms across the different IaC ecosystems, symptoms were found to be nearly uniformly spread. A *Chi-Square test* validated this observation, revealing no correlation between the type of symptom and the ecosystem it appears in.

4.2 RQ2: Bug Causes

Because of their distinct characteristics, the bug causes in the application layer may vary significantly from those in the runtime component (i.e., configuration units). Indeed, our analysis identified some root causes that are exclusive to IaC programs written in declarative DSLs. However, most of the root cause patterns are common across both configuration units and IaC programs. Figure 6 shows the distribution of the identified root cause categories, grouped based on the component in which the bug occurred. In the following, we provide descriptions and examples for every root cause category.

4.2.1 Common Bug Causes. Our analysis has pinpointed six underlying root causes prevalent in bugs within both the application layer and the runtime component of IaC systems: *System Interaction Bugs*, *Compatibility Issues*, *State Handling Bugs*, *Input Handling Bugs*, *Resilience bugs*, and *API-related Bugs*.

System Interaction Bugs: By definition, configuration units and IaC programs are not closed-world programs. Their fundamental purpose is to engage with the environment and induce a range of side effects. These interactions are commonly achieved through the execution of external shell commands, utilization of external libraries, or communication with REST APIs and remote services. *System interaction bugs* include cases where the interaction with the environment is not the intended one. This arises from (1) executing invalid or faulty commands (e.g., a command with incorrect arguments/command-line options, or escaping issues) (2) failing to execute crucial system operations (e.g., creation of important files, or installation of necessary packages), (3) carrying out unwanted or extraneous system operations. System interaction bugs are the most common ones: they appear in 87 out of 360 bugs (24%). They primarily affect configuration unit implementations (58/212), rather than IaC programs (29/148).

Example bug: ansible-collections/community.postgresql-314: To illustrate this root cause, consider an Ansible module (i.e., configuration unit) named `postgresql_info`, which is used to

gather information from PostgreSQL servers. To do so, the implementation of `postgresql_info` runs the PostgreSQL command `SHOW x`, which provides the current configuration of a setting `x` within a database server. However, a syntax error (*"syntax error at or near analyze"*) occurs when the module attempts to run `SHOW` on the setting `pg_partman_bgw.analyze`. Notably, this setting is available when a partition management extension is enabled in the underlying PostgreSQL server. The root cause of this syntax error is that "analyze" (included in the name of the setting) is a reserved keyword in PostgreSQL. The developers of `postgresql_info` fixed this bug by enclosing the arguments of `SHOW` with quotes: `SHOW "pg_partman_bgw.analyze"`.

Compatibility Bugs: Most of configuration units and IaC programs are designed to be platform and operating system independent. This means that they should work smoothly across various platforms and OSs. At the same time, they often utilize third-party software, such as databases and web servers, and are expected to handle different versions of these external entities effectively. A *compatibility issue* refers to a situation where a configuration unit or an IaC program is unable to operate as intended due to differences in the platforms, OSs, dependencies, or versions with which it interacts. This can occur when the code expects (1) a specific version or distribution of an operating system, or (2) a specific software package and dependency, but encounters a different version that leads to problematic executions. Out of the 360 studied bugs, 83 (23%) are due to compatibility issues, equally divided between configuration units and IaC programs.

Example bug: sous-chefs/vagrant-42 (compatibility issue with OS/platform): This bug is associated with a compatibility issue related to a specific OS/platform. There is a Chef resource called `vagrant_plugin`, managing Vagrant plugin installations. On Windows, its attempt to find the OS user's home directory using the Unix-specific Ruby function `Etc.getpwnam` causes a crash. The fix involves replacing `Etc.getpwnam` with the OS-agnostic `Dir.home`.

Example bug: voxpupuli/puppet-python-638 (compatibility issue with software dependency): This issue illustrates a compatibility problem in a Puppet program called puppet-python, which manages Python installations and related tools. In particular, this Puppet program contains a functionality for installing packages via `pip`, Python's package manager. Before installing a given package, the Puppet program checks whether the package is already installed in the system by running a specific `pip` command. However, this command always fails; thus making Puppet erroneously re-install the package at every run, breaking idempotency. The root cause stems from changes in `pip`'s dependency resolver starting from version 20.3. Pip versions between 20.3 and 21.4 require the use of a legacy resolver for compatibility. The fix involves the detection of the `pip` version; if it falls within the problematic range, the program switches to the legacy resolver by adding the `-use-deprecated=legacy-resolver` option when executing the `pip` check command.

State Handling Bugs: Configuration units are controllers and manipulators of the system state. They incorporate logic for assessing the system's current state, or identifying deviations between the current state and the desired one. In similar manner, IaC programs convert the system state into various formats for further processing (e.g., passing it as input to configuration unit invocations). A *state handling bug* occurs when a particular IaC code mishandles the system state. This is distinct from system interaction bugs, where the error lies in executing improper system operations. In state handling bugs, the code runs the correct operations but erroneously interprets or utilizes the results of those operations. This can happen because of one of the following scenarios. First, the buggy code has an inaccurate view of the system's current state. Second, there are flaws in the logic for detecting discrepancies between the current state and the desired one. Third, the buggy code incorrectly transforms the system state into wrong formats for further processing. State handling bugs are particularly common in configuration units (49/212), and highly rare in IaC programs (*only* 3/148). This is because configuration units *directly* manipulate the system's state.

In Section 4.1.4, we have described an issue whose root cause lies in system manipulation logic. Specifically, the `docker_container` Chef resource fails to identify deviations between the current system state and the desired one in a reliable manner. As a result, the buggy configuration unit restarts a Docker container in every run, violating idempotency.

Input Handling Bugs: Every configuration unit receives a set of arguments by the caller IaC program. Similarly, IaC programs expect a set of parameters by the user. Configuration units and IaC programs are responsible for validating these inputs and then processing, converting, or normalizing them as necessary. Many bugs appear when one of the aforementioned procedures is faulty, such as when a configuration unit fails to validate input correctly or lacks sufficient validation checks. Processing invalid inputs can compromise the entire IaC program execution. Input handling bugs account for 11% (40/360) of the studied bugs. Most of them (30/360) are found in configuration unit implementations.

Example bug: chef-boneyard/windows-424: Consider a Chef resource called `windows_path`, which manages the `PATH` environment variable in Windows. Calling this resource like so: `windows_path "C:/a/b"` ensures that the path `"C:/a/b"` is included in the `PATH` environment variable. However, there is a bug in how `windows_path` processes inputs containing forward slashes, e.g., `"C:/a/b"`. This is because in Windows the path separator is a backslash `"\"`, and not a forward slash `"/"`. The Chef resource fails to handle paths with forward slashes in a way that is compatible to Windows. As a result, the path `C:/a/b` is not properly interpreted when the `PATH` environment variable is later accessed. The developers fixed this bug by replacing all occurrences of `"/"` with `"\"` in the input path.

Resilience Bugs: Configuration units and IaC programs need to cope with the unpredictable behaviors that stem from their interactions with the external environment. Resilience refers to the ability of code to maintain its intended functionality and recover from errors or unexpected conditions in the system's environment. This means that despite the presence of unexpected disruptions, the code can handle these errors gracefully (e.g., via proper exception handling). *Resilience bugs* undermine the resilience property of configuration units and IaC programs, causing abnormal executions and abrupt terminations. 22 out of 360 instances (6%) are classified as resilience bugs.

To illustrate this root cause, consider again voxpupuli/puppet-rabbitmq-704 discussed in Section 4.1.3. In this example, a Puppet configuration unit interacts with a buggy RabbitMQ installation. When fetching the current version of RabbitMQ, the buggy RabbitMQ package returns an unexpected and strange output that contains invalid characters. This makes the Puppet configuration unit crash. The developers fixed this issue by handling such unpredictable disruptions gracefully.

API-related Bugs: Both configuration units and IaC programs come with an API that defines (1) a set of expected parameters along with their types, and (2) a set of expected return values. Developers consult the API documentation to interact with configuration units and IaC programs in a proper manner. *API inconsistency bugs* arise when the actual implementation does not comply with the documented API. API consistency bugs often introduce confusion to developers. We have found 19 API-related bugs in total (5%).

Example bug: ansible-collections/community.digitalocean-174: Consider the Ansible module `digital_ocean_vpc`, which creates or deletes virtual private cloud (VPC) networks in DigitalOcean cloud. According to its documentation, upon completion, this module should return a JSON that contains a key called `vpc`, which holds all the metadata information about the created/deleted VPC. Nevertheless, a flaw in the implementation of `digital_ocean_vpc` omits this key, making developers unable to access information about the managed VPC. The fix of the bug includes a new key in the output of the module with all the required information.

```
1  [Service]
2  EnvironmentFile=/etc/sysconfig/elasticsearch-es-01
3  User=elasticsearch
4  Group=elasticsearch
5  PIDFile=/var/run/elasticsearch/elasticsearch-es-01.
       pid
6  LimitMEMLOCK=
7  ...
```

(a) The generated `systemd` unit.

```
1  ...
2  <% if @memlock == 'unlimited' %>
3  LimitMEMLOCK=infinity
4  - <% else %>
5  + <% elsif @memlock %>
6  LimitMEMLOCK=<%= @memlock %>
7  <% end %>
8  ...
```

(b) Fixing a buggy template file.

Fig. 7. A *template bug*.

4.2.2 Issues in IaC programs. Figure 6b presents the distribution of root causes in IaC program bugs. Bugs in IaC programs are caused by a plethora of reasons. Besides the six root cause categories discussed in Section 4.2.1, there are four unique root cause categories pertinent only to IaC programs crafted in high-level DSLs. These include *Invalid DSL Bugs*, *Dependency Bugs*, *Template Bugs*, and *Bugs related to Hardcoded Values*.

Template Bugs: In addition to DSL code, IaC programs also contain template files written in specialized templating languages like Jinja2. The main goal of templates is to facilitate the generation of complex configuration files based on data provided by the users. Developers write text files that embed expressions that access IaC program variables to influence the file contents dynamically. In many situations, these template files use these expressions in a wrong manner, which eventually cause the creation of configuration files with undesired contents. We refer to these situations as *template bugs*. Template bugs lead to misconfigurations (Section 4.1.2) that significantly impact the operation of services, which in turn rely on the generated configuration files. We have classified 21 bugs as template bugs.

Example bug: puppet-elasticsearch-362: In this situation, a Puppet program managing Elasticsearch nodes generates a `systemd` unit file, where the property `LimitMEMLOCK=` is unset due to a faulty template (Figure 7a, line 6). This syntax error in the unit file prevents `systemd` from starting the Elasticsearch service. The fix involved omitting the `LimitMEMLOCK=` line from the unit file when the `memlock` Puppet variable is undefined (Figure 7b, lines 4, 5).

Bugs related to Hardcoded Values: Hardcoded configuration data often reside inside IaC programs. Such hardcoded data refer to various aspects of the system, such as software versions, file names, and URLs pointing to remote resources (e.g., upstream repositories). A *bug related to hardcoded values* occurs when one of those hardcoded values within an IaC program are incorrect. Incorrect hardcoded values are discovered in 17 cases.

Consider again voxpupuli/puppet-redis-425, initially described in Section 4.1.2. In this case, the Puppet program sets the mode of the Redis configuration file `/etc/redis/redis.conf`, using the hardcoded value `0644`. As a result, the file is readable by all local users when it should not. To fix this bug, the file permissions were changed to `0640`, making it accessible only for authorized users.

Dependency Bugs: In Section 2, we explained that IaC programs coordinate configuration units in a logical sequence. An important aspect of this orchestration is managing the execution order of inter-dependent configuration units. For example, consider an IaC program that invokes two configuration units: `A` and `B`. The former creates a file `myfile.txt`, and the latter consumes the same file. In this scenario, the configuration unit `A` should be invoked before `B`. This execution sequence is established using specific DSL features that define dependencies between configuration units. A *dependency bug* occurs when developers (1) fail

```
1  systemd::unit_file {"jira.service":
2    ensure => present,
3    content => epp("jira/jira.service.epp")
4  }
5  service {"jira": ensure => "running" }
```

Fig. 8. A Puppet program that contains a *dependency* bug.

to specify these essential ordering constraints, resulting in non-deterministic program executions, or (2) when they impose excessive restrictions, creating circular dependencies among the configuration units. In total, we have classified 14 instances as dependency bugs.

Example bug: voxpupuli/puppet-jira-315: Figure 8 shows a code snippet taken from a real-world Puppet program, which is used to manage a Jira installation. This Puppet program invokes two Puppet resources. The first resource is used to set up a `systemd` unit file for the Jira service based on the contents of a template (lines 1–4). The second resource called `service` ensures that the Jira service is operational (lines 5–7). The program exhibits a dependency bug in scenarios like altering the `JAVA_HOME` environment variable. Such a change prompts the `systemd::unit_file` resource to update the `systemd` unit file, reflecting the new Java environment for the Jira service. Despite the change in the service file, the `systemd` daemon does not restart. This leaves the Jira service running on the old version of Java. The root cause of the bug is that changes in the `systemd::unit_file` are not propagated to the `service` resource. To fix this issue, the developers introduced a dependency between `systemd::unit_file` and `service`. In this way, they can ensure that any modifications to the `systemd` file trigger a restart of the Jira service.

Invalid DSL Bugs: Every DSL employed by IaC systems comes with a set of specific syntactic and semantic rules that determine the validity of IaC programs. An example of such a semantic rule is that all IaC program variables should be initialized before they are used. A bug related to *invalid DSL* involves an IaC program violating these DSL rules, either syntactically or semantically. Invalid DSL programs usually lead to abrupt terminations, as the execution engine of the IaC system is unable to process the invalid code. Only 5 bugs have been classified as invalid DSL bugs.

Example bug: derdanne/puppet-nfs-38: In Puppet, every resource (configuration unit) within a Puppet program is uniquely identified by a combination of its type and name. The Puppet language does not allow the declaration of duplicate resources of the same type and name. However, a Puppet program named `puppet-nfs`, which is used to configure a Network File System (NFS), violates this property. Specifically, this Puppet program defines two conflicting `file` resources that manage the same underlying directory (e.g., `/mydir`). This causes the Puppet execution engine to reject the program and raise a corresponding error message.

4.2.3 Comparative Analysis. Consider again Figure 6a. We see that in most root cause categories, ecosystems follow similar trends. What is notable is that Ansible has the highest amount of *input handling bugs*, making up 18% (17/94) of the total, while Chef has the greatest percentage of *compatibility bugs* at 22% (17/76). Moving to IaC programs (Figure 6b), we observe that the majority of Ansible bugs occur due to *compatibility* or *system interaction* issues (21/26). Many Chef bugs are also caused due to these root causes (23/44), with a notable number (1) found in *templates* and (2) related to *hardcoded values* (15/44). Finally, Puppet bugs predominantly originate from *compatibility*, *template*, or *dependency* issues (41/78), with less prevalence in *system interaction* issues. We also performed separate *Chi-Square tests*, which confirm a significant difference between IaC ecosystems and root causes for bugs found in both IaC programs and configuration units.

4.3 RQ3: System State Requirements and Input Characteristics

Two primary factors affect the execution of IaC programs and configuration units: (1) the *user input* (desired state), and (2) the *current system state*. The state encapsulates any perceivable information about the system itself (e.g., operating system type), its execution status (e.g., running processes), and its expanded environment including all the other systems it interacts with. We examine how the current state and the user input affect the reproducibility of the studied bugs. System state requirements and user input characteristics can help devise strategies for bug reproduction (e.g., system state reconstruction) and new bug detection (e.g., automated input generation).

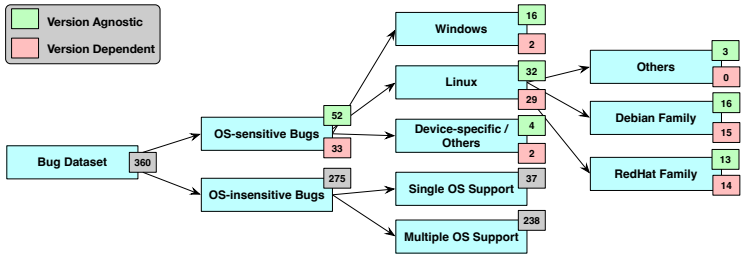


Fig. 9. Distribution of bugs in our dataset, categorized by OS sensitivity and family. OS-sensitive bugs are only reproducible in specific OS versions or only within a proper subset of the supported OSs. In contrast, OS-insensitive bugs are reproducible across all supported OS versions and are categorized based on the scope of the project's OS support, whether it is singular or multiple. The green box indicates the number of bugs that manifest in any OS version while the orange box counts the ones reproducible only on specific OS versions.

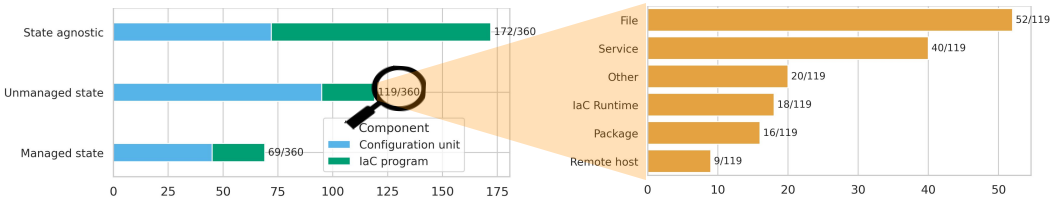


Fig. 10. Distribution of system state requirements: The left side of the chart categorizes bugs according to their system state requirements. The right side provides a detailed breakdown within the *unmanaged* state.

4.3.1 Operating Systems Requirements. Our study focuses on configuration-based IaC programs. Thus, the target systems are heterogeneous hosts running different operating systems. Specifically, 89% of the studied bugs (319/360) are found in IaC code that is designed to be *portable*, supporting multiple OSs and platforms. The IaC artifacts that we studied cumulatively support over 30 OSs, from standard and popular platforms, including Windows and Linux-based OSs, to specialized network OSs that empower network devices, such as Arista EOS. Only 41 out of 360 bugs (11%) are triggered in code that runs on a single OS and platform. For example, the Chef resource named `win_path` (Section 4.1) supports only Windows.

Our findings further indicate that a considerable portion (24% percent) of bugs are *OS-sensitive*. We define as *OS-sensitive* those bugs that (1) are triggered within a proper subset of the supported OSs of the target IaC program/configuration unit (*version agnostic*), or (2) manifest on a specific version of an OS, even when it is the only one supported by the IaC program / configuration unit (*version dependent*). For example, OS-sensitive bugs include cases where the IaC code runs flawlessly on Debian, but fails on RHEL. Out of 360 studied bugs, 85 are OS-sensitive, with 52 being *version agnostic* and 33 *version dependent*. This suggests that the identification of IaC bugs requires running the target programs on specific OS versions and distributions. OS-sensitive bugs are typically caused by *compatibility issues* (Section 4.2.1).

Standout operating systems: Figure 9 shows the distribution of OS-sensitive bugs in a tree format. Around 71% of the OS-sensitive bugs appear in the Debian/RedHat families, such as Ubuntu, CentOS, or RHEL, while there is a balance between *version agnostic* (32) and *version dependent* (29) bugs. On the contrary, running a multi-platform IaC program or configuration unit on Windows is enough to trigger many bugs (16). Note that only two of these bugs depend on specific versions. Out of the 275 OS-insensitive bugs, 37 of them appear in IaC artifacts that support a single OS.

4.3.2 State Reachability. Our bug analysis reveals that nearly half of the studied bugs (172 out of 360) are *state-agnostic*, that is, their manifestation depends *entirely* on the provided user input, and *not* on the current system state. State-agnostic bugs are generally easier to trigger and automatically search for, as they could be in theory detected using standard input fuzzers, such as AFL [M. Zalewski 2013]. However, the remaining bugs, surpassing half of the total (188/360), are *state-dependent bugs*. To trigger such bugs, a certain user input and a specific initial state are required. For example, a state-dependent bug may occur when running specific IaC code under input I and on an initial state S, but it is not triggered when the same code under I runs on a different initial state S'. We can differentiate these initial states between those that emerge from prior runs of the IaC program/configuration unit (*managed* state) and those that are out of its reach (*unmanaged* state).

Managed state: A managed state involves a number of system conditions that are the result of prior executions of an IaC program/configuration unit. For example, consider an IaC program named `A`. `A` will install a number of software packages on a computer, including the software package `p`. When `A` runs successfully, the *managed state* is the new state of the system with `p` installed. If there is a bug in `A` that manifests itself only after `p`'s installation, this bug depends on this specific state. We observed that a non-negligible (19%) amount of bugs stem from these managed states. Such states are highly important as they are reachable solely by running the IaC program/configuration unit, which can make them valuable for automated IaC testing. For example, the bug in the `docker_container` resource (sous-chefs/docker-1180—Section 4.1.4) is triggered only after a successful run of the same buggy Chef resource.

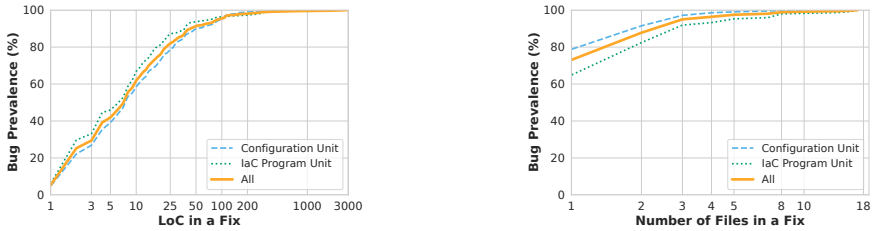
Unmanaged state: This refers to specific conditions within a system that are not established or influenced by the execution of an IaC program or its configuration units. Such states impact the program's behavior or outcomes even though the IaC code does not modify or manage these states directly. Unmanaged states are related to 33% (119/360) of studied bugs. Triggering these bugs necessitates a particular initial system state S unreachable via prior executions of the IaC program/configuration units. This is an important observation because we need techniques that not only invoke IaC code with diverse inputs, but also generate initial system states that look promising for triggering IaC bugs. Notably, such initial states can be potentially reached through the invocation of IaC program/configuration units other than the ones under test.

We have observed that some common system state requirements can be grouped into representative categories, each one with unique traits regarding reproduction and testing. Figure 10 shows their frequency. What is notable is that reproducing many of the state-dependent bugs requires the presence of certain services whose creation is not controlled by the buggy IaC program/configuration unit. For example, consider again ansible-collections/community.postgresql described in Section 4.2.1. In this example, the `postgresql_info` module reads the state of an existing PostgreSQL server without providing any functionality on how to create one. Still, the bug in `postgresql_info` is triggered exclusively in systems where a PostgreSQL server (service) with the partition management extension is active. Beyond services, other bugs are triggered by the presence of specific files, OS and PL-package versions, the IaC runtime itself (e.g., Ansible version, Python version), or remote hosts.

Table 3. The five most frequent input types appearing in the bug-triggering test cases

Data type	Occ (%)
Network (IP, port, firewall)	28%
File system (path, attrs)	19%
Package (name, version)	15%
Authentication (token, login info)	10%
Command (shell)	4%

4.3.3 Input Characteristics. Beyond its initial system state, the input of an IaC program/configuration unit also plays an important role in its runtime behavior. For completeness, we examined the bug-triggering test cases supplied in the bug reports and categorized the inputs into abstract data



(a) Cumulative distribution of lines of code in a fix. (b) Cumulative distribution of files in a fix.

Fig. 11. Size of bug fixes.

types. Table 3 presents the most prevalent ones. Note that we excluded common data types, such as, integers, or booleans. Interestingly, network-related inputs, such as IP addresses, ports, hostnames, and interfaces, are the most prevalent irrespective of the domain of the target component. Over one quarter of studied bugs are triggered by at least one such network-related input.

Another interesting observation is that IaC programs/configuration units often expose the API of the underlying resource they manage. For example, the API of a configuration unit that manages a PostgreSQL database might include a subset of PostgreSQL's API (e.g., a parameter named `dbname`). This one-to-one mapping could be useful to better understand the specific state affected by every input parameter of the IaC program/configuration unit.

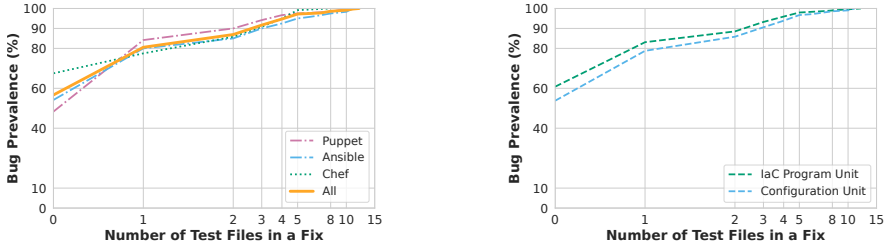
4.3.4 Comparative Analysis. With respect to OS requirements, we find that 17% (37/212) of bugs in configuration units are OS-sensitive, while the percentage nearly doubles to 32% (48/148) for IaC program bugs. When comparing OS sensitivity across different ecosystems, we found that Ansible bugs are predominantly OS-insensitive (87%), while nearly one third of Chef bugs (38/120) and one quarter of Puppet bugs (31/120) are OS-sensitive. Independent *Chi-Square tests* confirmed a significant association between OS sensitivity and both IaC components and ecosystems.

With respect to state reachability (Figure 10), the majority of IaC program bugs are state-agnostic (67%). In contrast, most of the bugs within configuration units are state-dependent (65%), and for the majority the required state is not managed by the given configuration unit (95/212). The system state they operate on can affect their behavior and make them susceptible to bugs. Note that this trend aligns with the overall architecture setup, because configuration units directly interact with the underlying system. Our observation regarding state-dependent bugs also indicates that configuration unit bugs are more difficult to reproduce when compared to bugs found in DSLs. To validate those observations we performed a *Chi-Square test*, which showed that configuration unit bugs and IaC program bugs are statistically different with respect to system state characteristics.

When comparing the state requirements between different ecosystems, we find that cases related to Ansible are mainly state-dependent (65%). On the other hand, state-agnostic bugs are most prevalent in Chef, accounting for 63% of its analyzed bugs. Finally, Puppet has an almost even distribution between state-dependent and state-agnostic bugs, 54% and 46%, respectively. To validate these cross-ecosystem comparisons, we conducted *Chi-Square tests*, which confirmed that for Ansible and Chef, state-dependent and state-agnostic bugs are statistically different. However, for Puppet, there is no significant difference, indicating both types of bugs are equally prevalent.

4.4 RQ4: Bug Fixes

4.4.1 Size of Bug Fixes. For every bug in our dataset, we extracted the corresponding fix and measured its size through automated means (see Section 3.2). In particular, our analysis focused on measuring both the number of files and the lines of source code affected by each fix, excluding those files and lines related to test code or documentation. Since every IaC package adheres to a



(a) Cumulative distribution of test files in a fix per IaC ecosystem.

(b) Cumulative distribution of test files in a fix per IaC component.

Fig. 12. Size of test files in fixes.

predefined directory structure, it was easy for us to identify whether the fix was associated with the runtime or the application layer (Section 2).

Figure 12 presents the cumulative distribution function for the lines of code (LoC) and the number of files modified per bug fix. Starting with LoC, the orange line in Figure 11a reveals that 95% of the fixes involve less than 100 LoC, with 39% requiring *fewer than five* LoC. The average number of lines modified per fix is 35, with a median of eight. These findings suggest that most fixes of IaC are relatively small, affecting a limited amount of code. With respect to the number of files modified by each bug fix, the orange line on Figure 11b indicates that fixes in IaC bugs touch few files: 95% of the bugs modify at most three files, while 72% modify exactly one file.

Comparative analysis: When comparing the fixes of (1) configuration units' implementations and (2) programs written in DSLs, fixes in DSL code generally involve fewer LoC (Figure 11a). Specifically, the median is seven LoC for fixes within the application layer, and eight LoC for fixes within configuration units. This difference is attributed to the nature of the fixes: IaC programs written in DSLs typically adjust a few declarations. On the contrary, configuration units, which require logic modifications in the source code, tend to involve more extensive changes. Running a Mann-Whitney U test on the distribution of the LoC sizes of IaC programs and configuration units confirms that the LoC difference is statistically significant.

With regards to the number of source files (Figure 11b), when comparing configuration units vs. IaC programs, we observe that fixes in configuration units tend to modify fewer files than the ones of high-level IaC programs. This finding contrasts with the finding of LoC (Figure 11a). The difference could be attributed to the fact that configuration units are typically implemented in single, yet large source files, while IaC programs are typically modular: their source code spans multiple files. Running a Mann-Whitney U test on the file distribution of the two components verifies that they are statistically different.

4.4.2 Size of Test Cases. Beyond source code, we decided to check whether developers introduce new tests along with their fixes to prevent potential regression bugs in the future. To do so, we examined the test files modified by each fix revision. As illustrated on Figure 12a, our analysis revealed that 57% do not modify any test files, indicating a potential area for improvement, particularly in Chef, where 68% of fixes lack test file updates. This trend suggests a preference for direct code modifications over test-driven development.

Furthermore, Figure 12b reveals that IaC programs tend to include fewer test cases in their fixes: 61% of IaC program bug fixes lack any test case modifications, compared to 54% for configuration units.

Table 4. An overview of papers and tools that provide bug-finding capabilities for both the application layer and the runtime. While numerous techniques focusing on the application layer, they often face challenges in reasoning about the dynamic behavior of IaC programs. This is because most of them rely on static analysis. The last column represents standard testing tools (e.g., unit and integration testing frameworks) employed in the development pipeline of IaC programs and configuration units. These tools are marked with half-filled circles, while useful, they offer limited functionality, as developers are required to manually write test cases and define test oracles. Notably, there is no paper focusing on bugs in configuration units (runtime).

	Bug		Static analysis/verification							Dynamic analysis		
		Trad Testing*	Puppeteer	PDG	SLIC	Glitch	GASEL	Rehearsal	SecureCode	Citac	FSMoVe	PRoTI
Application layer	System interaction bugs	◐	○	○	○	○	○	○	●	○	○	●
	Compatibility bugs	◐	○	○	○	○	○	○	○	○	○	○
	Template bugs	◐	○	○	○	○	○	○	○	○	○	●
	Hardcoded values	◐	●	○	●	●	●	○	○	○	○	●
	Dependency bugs	◐	○	○	○	○	○	●	○	●	●	○
	Input handling bugs	◐	○	○	○	○	○	○	○	○	○	●
	API-related bugs	◐	○	○	○	○	○	○	○	○	○	●
	Invalid DSL	◐	●	●	○	●	○	○	○	○	○	○
	Resilience bugs	◐	○	○	○	○	○	○	○	○	○	○
	State handling bugs	◐	○	○	○	○	○	○	○	○	○	●
Runtime	System interaction bugs	◐	○	○	○	○	○	○	○	○	○	○
	State handling bugs	◐	○	○	○	○	○	○	○	○	○	○
	Compatibility bugs	◐	○	○	○	○	○	○	○	○	○	○
	Input handling bugs	◐	○	○	○	○	○	○	○	○	○	○
	Resilience bugs	◐	○	○	○	○	○	○	○	○	○	○
	API-related bugs	◐	○	○	○	○	○	○	○	○	○	○

Table 5. The effectiveness of existing IaC test suites at identifying the bugs studied in our work. The failure to discover these bugs can be attributed to two main factors: (1) omission of tests by developers (red cells), or (2) inadequacy of existing tests to capture all relevant program behaviors (gray cells).

Step	Details	Bug count		
		Application layer	Runtime	Total
Bug breakdown	Bug dataset	148	212	360
	Tests improved during bug fix	58/360	98/360	156/360
	Bugs in repositories with no tests	3/360	4/360	7/360
	Tests were not improved during bug fix	87/360	110/360	197/360
Evaluation of test suites with unchanged files after bug fixes	Bugs fixed between 2019–2024 in tested IaC software	35/197	57/197	92/197
	Bugs with no component-specific tests	16/92	14/92	30/92
	Tests with compatibility issues	3/92	5/92	8/92
	Tests that ran successfully before bug fix	16/54	38/92	54/92

5 Discussion

We now present the status of the existing techniques for detecting bugs discussed in this work (Section 5.1). We also provide a number of implications and insights that stem from our research questions (Section 5.2).

5.1 Status of Existing Techniques

Table 4 consolidates existing practices and research focused on improving IaC reliability. The presence of filled circles indicates the capability of the described method to identify and eliminate the type of bug listed in the respective row. Half-circles indicate that while the technique can identify bugs, it does so without automation. Our table distinguishes between static-based techniques, dynamic-based techniques, and traditional testing. Traditional testing (column "Trad Testing") refers to all the unit testing and integration testing frameworks, such as RSpec [RSpec team 2024], `ansible-test` [Red Hat, Inc. 2024b], or Molecule [Red Hat, Inc. 2024a], which are available to developers during the development process of IaC programs and configuration units.

Traditional testing: In traditional testing, IaC developers test the behavior of their programs and configuration units via manual inputs and assertions. This process may also involve setting

up or mocking the initial system state on which the software under test operates. Even though traditional testing can *theoretically* capture all types of bugs studied in this work, it often lacks automation. This is because developers need to manually craft inputs and states that capture all program behaviors. Therefore, some edge cases might not be covered by the existing tests.

To back up the aforementioned statement, we further assessed the extent to which the existing test suites of IaC developers could capture the bugs studied in our work. Out of the 360 selected bugs (as detailed in Table 5), we first detected that in 156 cases, IaC developers not only fixed their bugs but also improved the corresponding test suites to prevent future regressions (Section 4.4.2). This indicates that the original tests were inadequate to capture these 156 bugs. In 7/360 cases, bugs were found in GitHub repositories with no tests at all. For the remaining 197 bugs, which were found in repositories with existing tests but did not involve any changes to test-related files during the fix, we conducted a small-scale experiment. This experiment focused on the 92 bugs fixed between 2019 and 2024, as detailed in Table 5. We excluded 30 bugs, which were found in repositories that although contained test files, there were no tests related to the buggy configuration unit or IaC program. We further excluded 8 bugs whose tests could not be executed because of compatibility issues, e.g., reliance on an outdated version of Ruby that is no longer supported by recent operating systems. Thus, we were left to work with 54 bugs from projects that had tests, but these tests were neither updated nor improved during the bug fix. For each of these cases, we checked out the repository version *immediately before* the merge commit of the fix and manually executed the unit tests of the buggy configuration unit or IaC program, following the testing guidelines of every project. The tests passed in *all* these instances, indicating that the existing tests were incapable of detecting the buggy behavior. We also measured the code coverage of these successful tests. On average, the test suites achieved 50% code coverage.

In summary, the above results indicate that IaC developers missed the bugs in at least 247/360 instances. This occurred because (1) developers did not write tests (37/360— red cells of Table 5), or (2) the existing test cases missed important program behaviors (210/360— gray cells of Table 5). To overcome the limitations of traditional testing, research on IaC has explored various techniques (both static and dynamic) to improve bug detection. Below, we distinguish these techniques based on their scope (application layer vs. runtime).

Application layer: The focus of all existing papers has been the application layer. We divide these papers into static and dynamic techniques. *Puppeteer* [Sharma et al. 2016] is a rule-based code smell detector for the Puppet DSL. The current implementation of Puppeteer primarily focuses on issues related to code style (e.g., improper alignment). Puppeteer can be extended with more rules to catch a wider array of bugs, especially those within the *invalid DSL* and *bugs related to hardcoded values* categories. *PDG* [Opdebeeck et al. 2022] is another code smell detector that relies on the concept of program dependence graphs, a representation that captures the control and data flow of Ansible roles. Based on this representation, PDG detects bugs related to *invalid DSL*, with particular focus on smelly Ansible variables (e.g., suspicious variable overriding). *SLIC* [Rahman et al. 2019], *Glitch* [Saavedra and Ferreira 2023; Saavedra et al. 2023], and *GASEL* [Opdebeeck et al. 2023b] are all static analysis tools for security smell detection in IaC programs. Such smells stem from incorrect usage of *hardcoded values* in IaC scripts, e.g., the use of hardcoded passwords, or creating files with unsafe permissions (see voxpupuli/puppet-redis-425). Beyond security smells, *Glitch*'s internal representation is also useful for general code smell detection. *Rehearsal* [Shambaugh et al. 2016] employs static verification for Puppet programs. It models certain Puppet resources (e.g., `file`), and formally defines the properties of determinism and idempotency. Then it checks if these properties are violated to identify potential *dependency bugs*. *SecureCode* [Dai et al. 2020] extracts the invocation of *shell* scripts from a given IaC program, and then analyzes them using existing linters to detect unreliable *system interactions*.

Although static analysis and verification are effective at identifying bugs such as *invalid DSL*, or *bugs related to hardcoded values*, they have inherent limitations when it comes to deeper issues. This is because static analysis is unable to reason about the dynamic behavior of IaC programs and their interaction with external entities. For example, *Rehearsal* cannot handle realistic Puppet programs that invoke the `exec` Puppet resource, which is used to run arbitrary shell scripts.

Dynamic analysis techniques for IaC programs include *Citac* [Hanappi et al. 2016], which executes a given Puppet or Chef program by invoking the enclosing configuration units in several possible execution orders. The goal is to uncover non-deterministic executions caused by *dependency bugs*. *FSMoVe* [Sotiropoulos et al. 2020] also detects *dependency bugs*, but using a different approach. Specifically, it employs system call tracing to identify the side effects (e.g., files read/written) of Puppet resources invoked by a program execution. Then, based on the system call traces, it infers dependency relationships between Puppet resources, and cross-checks these relationships against the dependencies declared by developers in the Puppet DSL. *ProTI* [Sokolowski and Salvaneschi 2023; Sokolowski et al. 2024] is used to test IaC programs written in Pulumi [Pulumi 2024], which is a provision-based IaC system. To do so, ProTI employs an approach called *Automated Configuration Testing*, which relies on property-based unit testing. ProTI mocks the behavior of configuration units by generating synthetic objects that represent their outputs. These mock outputs (which reflect the system's state) enable ProTI to test how IaC programs process these states, thus uncovering *state handling bugs*. Furthermore, ProTI supports the development of test oracle plugins, which verify the interaction of the caller IaC program with the underlying configuration units by checking the correctness of the arguments passed to the mocked configuration units (*system interaction bugs*).

Runtime: Table 4 shows that there is a significant gap in bug-finding techniques for configuration unit implementations written in traditional programming languages. All the aforementioned methods either operate on IaC DSLs or reason about how different configuration unit invocations within an IaC program are combined and structured. Currently, only traditional methods like unit testing and integration testing are employed to test the implementations of configuration units. However, as already discussed (Table 5), traditional testing might fail to capture some bugs. The challenge in validating IaC software via unit testing or integration testing lies in its complex interactions with external environments and the diverse states it can encounter, which are difficult for developers to fully predict in their hand-written tests. Therefore, better automated testing approaches are needed to effectively test IaC software behaviors under interesting inputs and states. Another factor is testability: many configuration units are hard to test, as they require complex system states, such as combinations of files, services, or package versions. We believe that our research offers valuable insights for creating targeted approaches to enhance the reliability of configuration units (see Section 5.2).

5.2 Lessons Learned and Takeaways

IaC bugs exhibit a variety of symptoms. Our symptom analysis (Section 4.1) reveals that IaC bugs may affect system components in diverse ways, from program crashes to operational issues. To effectively capture IaC bugs, testing methods should incorporate diverse test oracles. For example, to catch *external configuration failures* and *internal errors*, the test oracle could verify the exit code of the IaC program under test (non-zero exit code reveals unexpected program failures). Interestingly, such a simple test oracle could capture 61% of the bugs in our dataset. For *misconfigurations*, which are often experienced by users through service failures, a test oracle could monitor service health and logs for anomalies. That said, some IaC bugs might still necessitate domain-specific test oracles, like verifying OS installations for version inconsistencies, network configuration, and more.

IaC bug root causes mainly relate to system interaction and system state manipulation, as highlighted in Figure 6. Together, *system interaction*, *state handling*, and *resilience bugs* constitute

about 45% of studied bugs. For configuration units, such bugs are even more prevalent, as 125 out of the 212 configuration unit bugs fall within these categories (59%). Subsequently, future research should focus on targeted testing methods that exercise the aforementioned buggy procedures within IaC code, e.g., employing fault injection techniques [Banabic and Candea 2012; Chen et al. 2023; Marinescu et al. 2010] to address *resilience bugs*. In the case of *system interaction bugs*, a mitigation technique could involve a `check` mode that enables users to verify the validity of the underlying external commands invoked by configuration units. Such a mode could help detect the syntax error generated by the `SHOW` command of the ansible-collections/community.postgresql module (Section 4.2.1). Another way to address *system interaction bugs* could involve a system-aware verification method inspired by the work of Sun et al. [2024] on verifying Kubernetes controllers. Such a verification method could capture and enforce system state invariants, thus proving the correctness of a program under specific system state conditions.

IaC code suffers from compatibility issues. Approximately 23% of studied bugs are caused by compatibility issues with OSs, platforms, or software dependencies. This suggests that IaC programmers should allocate time to thoroughly test their programs across different OSs and dependency versions. Surprisingly, despite the high prevalence of such bugs, there are currently no automated techniques specifically designed to detect compatibility bugs, neither in IaC programs nor configuration units (See Table 4). To fill this gap, inspired by similar techniques applied in Android applications [Fazzini and Orso 2017; Sun et al. 2021], a cross-platform, cross-dependency, testing approach could help automatically identify inconsistencies across different OS or software versions (e.g., see *Compatibility issues* in Section 4.2.1) versions. Additionally, future empirical studies can further assess the prevalence and the evolution of compatibility issues by conducting larger scale, deeper analyses within IaC ecosystems.

Fixes of IaC bugs are small and touch few source files. Figure 12 shows that bug fixes require small modifications that span few or even a single source file. This makes automated program repair techniques promising for IaC code. Many IaC bug fixes are formulaic, indicating that automating these fixes could be feasible. For example, fixing *dependency bugs* just requires the declaration of a missing dependency link between two configuration unit invocations. Fixing *system interaction bugs* due to missing system operations (e.g., missing OS packages or directory) often requires the creation of the missing elements. While initial automated repair methods have been applied to IaC programs [Weiss et al. 2017], the effectiveness of such techniques for configuration units, which are written in imperative code, is yet to be determined.

IaC bugs are hard to reproduce. More than half (52%) of the bugs are state-dependent (Figure 10), as they only manifest under specific system conditions. This contrasts with traditional testing methods focused on generating varied user inputs [M. Zalewski 2013]. Instead, an effective testing approach for IaC should simulate diverse and interesting initial system states. In this case, a key technical challenge is the complexity of the target system, which can contain abundant resources. Such resources may range from simple files to more complex and domain-specific entities, (e.g., database tables). Therefore, future testing strategies need to pinpoint which system resources impact IaC execution and devise methods to manipulate these resources (e.g., updating file contents) to explore and test new initial system states. For example, previous research [Sotiropoulos et al. 2020] indicates that we can learn the set of the affected system resources by monitoring an IaC program's system calls. With that knowledge, we can meaningfully mutate these resources during testing.

Alternatively, future research could focus on adapting Kubernetes-based testing tools such as Sieve [Sun et al. 2022] and Acto [Gu et al. 2023], which are designed to test Kubernetes controllers/operators by generating new inputs and states (Section 2.1). However, adapting these tools to IaC presents a significant research challenge because unlike Kubernetes controllers/operators, IaC

interacts with a broader range of APIs. This requires methods capable of inferring (1) properties, (2) constraints, and (3) semantics from arbitrary APIs, not just those specific to Kubernetes.

IaC code receives inputs with domain-specific types. Table 3 shows that IaC programs frequently expect diverse input types including file paths and package names. The challenge here is designing generators that produce *valid* inputs that incorporate domain knowledge, such as IPs, Docker image names, etc. A worthwhile option is to explore whether generative AI methods (e.g., a ChatGPT prompt) can produce valid inputs given the API documentation of IaC code or the description of the corresponding data types.

Tooling around configuration units is limited. Table 4 shows a notable gap: there are no automated techniques for validating the implementations of configuration units. This can be attributed to the fact that, in contrast to bugs in IaC Programs, the majority of bugs in configuration units are state-dependent, and therefore harder to reproduce (Figure 10). Yet, configuration units are extremely important for the overall reliability of IaC. A bug in a single configuration unit propagates to all IaC programs that utilize it, which in turn render production systems and deployments unreliable. Therefore, we suggest that future research should build targeted reliability and verification tools for configuration units.

6 Related Work

Understanding bugs in IaC: There are defect analyses in IaC that are close to our work. Rahman et al. [2020] collected and manually analyzed 1,448 defect-related commits from 61 open-source Puppet programs. Their analysis considered the description of the selected commits, along with the description of any linked bug reports. They proposed eight categories of IaC defects. Some of their key results indicate that bugs related to configuration data are the most common ones, while idempotency issues make up the least frequent category. However, their analysis did not consider defects in configuration units or investigate the ways these defects are triggered or manifest themselves.

In another recent study, Hassan et al. [2024] examined 5,110 state reconciliation defects from Ansible modules, and grouped them into eight categories, namely, *auxiliary*, *conditional*, *idempotency*, *inventory*, *security*, *state inquiry*, *state regulation*, *type*. State reconciliation defects correspond to the *state handling bugs* of our work. Therefore, their study is a subset of ours since they examined state handling bugs from the main Ansible repository, excluding roles. In contrast, our study encompasses a broader range of bugs across multiple dimensions: symptoms, root causes, and trigger conditions. Overall, the work of Hassan et al. [2024] provides valuable insights into specific functionalities that are prone to state reconciliation bugs, such as access control, computing clusters, and caching mechanisms, likely due to the nature of the modules they studied (mostly security- and cloud-oriented modules). Our work complements the above by offering a broader, multifaceted analysis including a wider range of modules. Notably, we did not analyze any bugs studied by Hassan et al. [2024].

In a separate study, Goodwin et al. [2023] examined and analyzed bugs in Knative runtimes. Knative [Knative 2019] is an open-source platform for the development, deployment, and management of modern serverless and event-driven applications on Kubernetes. Many Knative bugs share similar root causes with IaC bugs. For example, some Knative bugs are caused by incorrect logic when (1) querying the health status and readiness of Knative resources (*system handling bugs*), or (2) interacting with Kubernetes to manage the resources required for the application deployment (*system interaction bugs*). However, unlike the IaC bugs examined in our study, system handling and system interaction bugs in Knative runtimes mainly arise due to race conditions and unexpected interleavings between Knative and Kubernetes components. Finally, many Knative

bugs have distinct root causes unrelated to IaC, such as autoscaling, request routing, or semantics of serverless function versions.

Comparison with existing studies: The objective of our paper is to pave the way for the *purposeful* development of effective fault detection techniques for IaC programs and configuration units. Our analysis is centered around three important dimensions:

- *Identification of symptoms*: Helps address the test oracle problem [Weyuker 1982], making it easier to decide whether a specific IaC software behavior is problematic or expected.
- *Identification of root causes*: Helps identify the faulty procedures within IaC programs and configuration units, and design targeted solutions that stress-test those faulty components.
- *Identification of trigger conditions (system state and input characteristics)*: Helps design techniques that generate inputs and states that are likely to trigger IaC bugs.

All these factors make our work distinct from the aforementioned studies on IaC and other platforms. Our work is the first to address all these questions specifically in the context of IaC, providing valuable insights for the effective design of fault detection and localization tools (Section 5.2).

Other empirical studies on IaC: There have been numerous empirical studies on IaC, focusing on quality and security aspects. Initial studies started with interviews that highlighted practitioners' views on the challenges of ensuring IaC quality [Guerriero et al. 2019]. Subsequent research introduced specific metrics and models for quality assessment. For example, Van der Bent et al. [2018] presented a quality model assessing the maintainability of Puppet programs. The proposed model was validated by Puppet programmers from the industry. In a similar way, Dalla Palma et al. [2020] suggested a catalog of 46 metrics, (e.g., number of files, size of comments) for characterizing the quality of Ansible programs. Different studies have applied their own quality metrics and models, revealing that many IaC programs exhibit issues related to code smells, such as variable shadowing [Opdebeeck et al. 2022; Sharma et al. 2016].

Researchers have also explored the security aspect of IaC programs. Starting with Puppet [Rahman et al. 2019], and later with Ansible and Chef [Rahman et al. 2021], research has shown that the occurrence of security smells is pervasive in IaC programs. Their proposed list of security smells includes seven issues, such as the use of hardcoded passwords, or the use of insecure HTTP connections. As we show in Section 5.1 the findings of these studies have enabled the development of various techniques for security smell detection. Recently, the focus has gradually shifted to studying security smells in the Kubernetes ecosystem [Rahman et al. 2023]. Our study enriches existing research by offering the first in-depth analysis of IaC bugs.

7 Conclusion

We presented the first comprehensive analysis of bugs in well-established Infrastructure as Code (IaC) ecosystems. Our bug examination has uncovered several insights regarding the nature of these bugs, their symptoms, causes, trigger conditions, and fixes. IaC bugs primarily lead to abrupt program terminations, or system misconfigurations that ultimately result in service failures and outages. Correctness issues found in the system interaction and state management logic are responsible for the majority of studied bugs. Reproducing these bugs is challenging: half of them are triggered only under certain initial states, such as the presence of files/packages, while 24% bugs are OS-sensitive. Based on our findings, we summarized the-state-of-the-art techniques for IaC reliability, identified their limitations and gaps, and discussed several implications and future directions. For example, unlike traditional programs that rely solely on user input, future testing techniques for IaC should exercise code under different initial system states and environments. We hope that our work will be invaluable for researchers and practitioners of IaC to better improve the reliability and robustness of computing infrastructures.

Data-Availability Statement

The research artifact [Drosos et al. 2024] is available at Zenodo under the MIT license. It provides the scripts, the data, and the results presented in this study. It also offers guidelines on how to apply our methodology to other IaC ecosystems (e.g., Terraform).

Acknowledgments

We thank the anonymous OOPSLA reviewers for their constructive comments. We also thank Panos Louridas for his assistance with the statistical tests.

References

Amazon Web Services, Inc. or its affiliates. 2017. *Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region.* https://aws.amazon.com/message/41926/ [Online; accessed 21-February-2024].

Ansible. 2024a. *Ansible Galaxy.* https://galaxy.ansible.com/ui/ [Online; accessed 21-February-2024].

Ansible. 2024b. *Ansible Playbook Content Organisation.* https://docs.ansible.com/ansible/2.8/user_guide/playbooks_best_practices.html#content-organization [Online; accessed 28-February-2024].

Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. 2017. DevOps: Introducing Infrastructure-as-Code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 497–498. https://doi.org/10.1109/ICSE-C.2017.162

Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. 2020. Actor concurrency bugs: a comprehensive study on symptoms, root causes, API usages, and differences. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 214 (nov 2020), 32 pages. https://doi.org/10.1145/3428282

Radu Banabic and George Candea. 2012. Fast black-box testing of system recovery code. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 281–294. https://doi.org/10.1145/2168836.2168865

Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-typed programs can go wrong: a study of typing-related bugs in JVM compilers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 123 (oct 2021), 30 pages. https://doi.org/10.1145/3485500

Chef Software, Inc. 2024a. *Chef Cookbook Directory Structure.* https://docs.chef.io/cookbook_repo/#cookbook-directory-structure [Online; accessed 28-February-2024].

Chef Software, Inc. 2024b. *The source of Chef cookbooks - Chef Supermarket.* https://supermarket.chef.io/ [Online; accessed 21-February-2024].

Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. 2023. Push-Button Reliability Testing for Cloud-Backed Applications with Rainmaker. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1701–1716. https://www.usenix.org/conference/nsdi23/presentation/chen-yinfang

Alva Couch and Yizhan Sun. 2003. On the Algebraic Structure of Convergence. In *Self-Managing Distributed Systems*, Marcus Brunner and Alexander Keller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 28–40.

Ting Dai, Alexei Karve, Grzegorz Koper, and Sai Zeng. 2020. Automatically detecting risky scripts in infrastructure code. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 358–371. https://doi.org/10.1145/3419111.3421303

Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. 2020. Toward a catalog of software quality metrics for infrastructure code. *Journal of Systems and Software* 170 (2020), 110726. https://doi.org/10.1016/j.jss.2020.110726

Thomas Delaet, Wouter Joosen, and Bart Vanbrabant. 2010. A survey of system configuration tools. In *Proceedings of the 24th International Conference on Large Installation System Administration* (San Jose, CA) *(LISA'10)*. USENIX Association, USA, 1–8.

Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A Comprehensive Study of Real-World Numerical Bug Characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 509–519. https://doi.org/10.1109/ASE.2017.8115662

Georgios-Petros Drosos, Thodoris Sotiropoulos, Georgios Alexopoulos, Dimitris Mitropoulos, and Zhendong Su. 2024. *Artifact for OOPSLA 2024 paper: "When Your Infrastructure is a Buggy Program: Understanding Faults in Infrastructure as Code Ecosystems".* https://doi.org/10.5281/zenodo.12668895

Aryaz Eghbali and Michael Pradel. 2021. No strings attached: an empirical study of string-related software bugs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 956–967. https://doi.org/10.1145/3324884.3416576

Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 308–318.

GitHub, Inc. 2014. *DNS Outage Post Mortem.* https://github.blog/2014-01-18-dns-outage-post-mortem/ [Online; accessed 21-February-2024].

Tim Goodwin, Andrew Quinn, and Lindsey Kuper. 2023. What goes wrong in serverless runtimes? A survey of bugs in Knative Serving. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies* (Rome, Italy) *(SESAME '23)*. Association for Computing Machinery, New York, NY, USA, 12–18. https://doi.org/10.1145/3592533.3592806

Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. 2023. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 96–112. https://doi.org/10.1145/3600006.3613161

MIchele Guerriero, Martin Garriga, Damian A. Tamburri, and Fabio Palomba. 2019. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 580–589. https://doi.org/10.1109/ICSME.2019.00092

Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. 2016. Asserting reliable convergence for configuration management scripts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/2983990.2984000

HashiCorp. 2024. *Automate infrastructure on any cloud with Terraform.* https://www.terraform.io/ [Online; accessed 21-February-2024].

Md Mahadi Hassan, John Salvador, Shubhra Kanti Karmaker Santu, and Akond Rahman. 2024. State Reconciliation Defects in Infrastructure as Code. *Proc. ACM Softw. Eng.* 1, FSE, Article 83 (jul 2024), 24 pages. https://doi.org/10.1145/3660790

Knative. 2019. Knative Documentation. https://knative.dev Accessed: 2024-08-03.

Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 517–530. https://doi.org/10.1145/2872362.2872374

Julien Lepiller, Ruzica Piskac, Martin Schäf, and Mark Santolucito. 2021. Analyzing Infrastructure as Code to Prevent Intra-update Sniping Vulnerabilities. In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 105–123.

M. Zalewski. 2013. American fuzzy lop. https://lcamtuf.coredump.cx/afl/. Online accessed; 05-08-2021.

Paul D. Marinescu, Radu Banabic, and George Candea. 2010. An extensible technique for high-precision testing of recovery code. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 23.

Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. 2019. Casting about in the Dark: An Empirical Study of Cast Operations in Java Programs. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 158 (Oct. 2019), 31 pages. https://doi.org/10.1145/3360584

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

Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2022. Smelly variables in Ansible infrastructure code: detection, prevalence, and lifetime. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/3524842.3527964

Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2023a. Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 534–545. https://doi.org/10.1109/MSR59073.2023.00079

Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2023b. Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 534–545. https://doi.org/10.1109/MSR59073.2023.00079

Perforce. 2024. *Puppet Infrastructure & IT Automation at Scale.* https://www.puppet.com/ [Online; accessed 21-February-2024].

Progress. 2024. *Chef Software DevOps Automation Solutions.* https://www.chef.io/ [Online; accessed 21-February-2024].

Pulumi. 2024. *Pulumi - Infrastructure as Code in any programming language.* https://www.pulumi.com/ [Online; accessed 21-February-2024].

Puppet. 2024a. *Puppet Forge.* https://forge.puppet.com/ [Online; accessed 21-February-2024].

Puppet. 2024b. *Understanding the Puppet Directory.* https://www.puppet.com/blog/puppet-directory [Online; accessed 28-February-2024].

Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. 2020. Gang of eight: a defect taxonomy for Infrastructure as Code scripts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 752–764. https://doi.org/10.1145/3377811.3380409

Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: security smells in Infrastructure as Code scripts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 164–175. https://doi.org/10.1109/ICSE.2019.00033

Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. 2021. Security Smells in Ansible and Chef Scripts: A Replication Study. *ACM Trans. Softw. Eng. Methodol.* 30, 1, Article 3 (jan 2021), 31 pages. https://doi.org/10.1145/3408897

Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. 2023. Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 99 (may 2023), 36 pages. https://doi.org/10.1145/3579639

Red Hat, Inc. 2024a. *Ansible Molecule*. https://ansible.readthedocs.io/projects/molecule/ [Online; accessed 01-August-2024].

Red Hat, Inc. 2024b. *Testing Ansible*. https://docs.ansible.com/ansible/latest/dev_guide/testing.html [Online; accessed 01-August-2024].

RedHat, Inc. 2024. *Ansible is simple IT automation*. https://www.ansible.com/ [Online; accessed 21-February-2024].

RSpec team. 2024. *RSpec: Behaviour Driven Development for Ruby*. https://rspec.info/ [Online; accessed 01-August-2024].

Nuno Saavedra and João F. Ferreira. 2023. GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (, Rochester, MI, USA,) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 47, 12 pages. https://doi.org/10.1145/3551349.3556945

Nuno Saavedra, João Gonçalves, Miguel Henriques, João F. Ferreira, and Alexandra Mendes. 2023. Polyglot Code Smell Detection for Infrastructure as Code with GLITCH. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2042–2045. https://doi.org/10.1109/ASE56229.2023.00162

Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: a configuration verification tool for Puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 416–430. https://doi.org/10.1145/2908080.2908083

Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/2901739.2901761

Daniel Sokolowski and Guido Salvaneschi. 2023. Towards Reliable Infrastructure as Code. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. 318–321. https://doi.org/10.1109/ICSA-C57050.2023.00072

Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. 2024. Automated Infrastructure as Code Program Testing. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1585–1599. https://doi.org/10.1109/TSE.2024.3393070

Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. Practical fault detection in Puppet programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 26–37. https://doi.org/10.1145/3377811.3380384

Diomidis Spinellis. 2012. Don't Install Software by Hand. *IEEE Software* 29, 4 (2012), 86–87. https://doi.org/10.1109/MS.2012.85

Stack Exchange Inc. 2023. *2023 Developer Survey*. https://survey.stackoverflow.co/2023/#other-tools [Online; accessed 01-April-2024].

Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and finding system setting-related defects in Android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 204–215. https://doi.org/10.1145/3460319.3464806

Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing For Cluster Management Controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 143–159. https://www.usenix.org/conference/osdi22/presentation/sun

Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: Verifying Liveness of Cluster Management Controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 649–666. https://www.usenix.org/conference/osdi24/presentation/sun-xudong

The Kubernetes authors. 2024. Kubernetes: Production-Grade Container Orchestration. Kubernetes Homepage. Retrieved from http://kubernetes.io/.

Eduard Van der Bent, J. Hage, Joost Visser, and Georgios Gousios. 2018. How good is your Puppet? An empirically defined and validated quality model for Puppet. In *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2018)*. IEEE, United States, 164–174. https://doi.org/10.1109/SANER.2018.8330206 25th IEEE International Conference on Software Analysis, Evolution and Reengineering 2018, SAMER 2018 ; Conference date:

20-03-2018 Through 23-03-2018.

Joost Visser, Sylvan Rigal, Gijs Wijnholds, and Zeeger Lubsen. 2016. *Building software teams: Ten best practices for effective software development.* " O'Reilly Media, Inc.".

Aaron Weiss, Arjun Guha, and Yuriy Brun. 2017. Tortoise: interactive system configuration repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 625–636.

Elaine J Weyuker. 1982. On testing non-testable programs. *Comput. J.* 25, 4 (1982), 465–470. https://doi.org/10.1093/comjnl/25.4.465

Wikimedia Commons. 2017. *Incident documentation/20170118-Labs.* https://wikitech.wikimedia.org/wiki/Incidents/2017-01-18_Labs [Online; accessed 21-February-2024].

Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (, Seattle, WA, USA,) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1319–1331. https://doi.org/10.1145/3597926.3598138

Received 2024-04-05; accepted 2024-08-18