Lessons from Research to Practice on Writing Better Quality Puppet Scripts

Akond Rahman
Department of Computer Science
Tennessee Tech University
Cookeville, TN, USA
arahman@tntech.edu

Tushar Sharma
Department of Computer Science
Dalhouise University
Nova Scotia, Canada
tushar@dal.ca

***Abstract*—Infrastructure as Code (IaC) scripts, such as Puppet scripts, provide practitioners the opportunity to provision computing infrastructure automatically at scale. Poorly written IaC scripts impact various facets of quality (such as security and maintainability) and, in turn, may lead to serious consequences. Many of the ill-effects can be avoided or rectified easily by following recommendations derived from research and best practices gleaned from experience. While researchers have investigated methods to improve quality aspects of Puppet scripts, such research needs to be summarized and synthesized for industry practitioners. In this article, we summarize recent research in the IaC domain by discussing key quality issues, specifically security and maintainability smells, that may arise in an IaC script. We also mine open-source repositories from three organizations (Mozilla, Openstack, and Wikimedia) and report our observations on the identified smells. Furthermore, we also synthesize recommendations from the literature for software practitioners that could improve the quality of IaC scripts. Software development teams dealing with large computing infrastructure can get benefited from the actionable recommended practices. In addition, researchers in the domain may use this study to find opportunities to improve the state-of-the-art.**

***Index Terms*—puppet, code smell, infrastructure as code, devops, quality, empirical study, bugs, security**

I. Introduction

Infrastructure as code (IaC) is the practice automating system deployment and specifying system configurations through code [12], [16]. IaC emphasizes the need of treating configuration code similar to production code and thus advocates adopting good software engineering practices, such as testing, static analysis, and version control for configuration code. With the high adoption of IaC tools, such as Puppet, software development teams today maintain a considerable size of code written for system deployment and configuration. It highlights the need to analyze, understand, and maintain the quality of IaC scripts.

Researchers have investigated various aspects of IaC frameworks and tools to better understand the needs of practitioners and improve their artifacts and processes [3]. Specifically, on the quality side, researchers investigated defect categories [10], [15], [18], [19], semantics and code properties [8], [14], and maintainability [3], [6], [15], [16] of configuration code. However, often, software engineering research and practice exhibit a disconnect, which hinders practitioners to benefit from the state-of-the-art research efforts conducted by the research community [1], [2]. Research in the IaC field also suffer from this limitation, and therefore, needs to be systematically summarized and disseminated. Such summarization can be helpful for (i) *practitioners*, who can learn from the findings to improve the quality of IaC scripts, and (ii) *researchers*, who can get a concise overview of the state-of-the-art research related to IaC to further build new methods and techniques.

To that end, in this paper, we summarize the findings from recent literature on the quality of IaC scripts and synthesize recommendations for software practitioners. In the next sections, we first describe the employed methodology and then explored a few research questions based on the goal of our study, explained in consecutive sections.

II. Methodology

The goal of this paper is to summarize and synthesize Puppet-related research for practitioners so that they can leverage the benefits from the state-of-the-art to improve the quality of their IaC scripts. Towards that goal, we conduct a literature review of publications that discuss quality aspects for Puppet scripts. We structure our paper addressing the following three research questions.

- **RQ1.** What quality issues can arise in Puppet scripts?
- **RQ2.** How frequently do quality issues occur in Puppet scripts?
- **RQ3.** What activities practitioners may adopt to mitigate quality concerns in Puppet scripts?

A. Literature search

We use three databases to identify publications that are related to Puppet quality: Google Scholar, IEEE Xplore, and the ACM Digital Library. We use the search string 'quality of Puppet scripts'. In the case of each scholar database, we collect the first 250 search results sorted based on relevance, as determined by all the searched digital libraries. We apply the following inclusion criteria.

- The publication must discuss Puppet-related research in the context of IaC. This allows us to filter irrelevant results, such as publications that focus on puppets used for entertainment;

- The publication was published on or after 2010. We use this year as Puppet was first released in the year of 2009;
- The publication is not a duplicate of another; and
- The publication discusses quality aspects of Puppet-related development.

We also adopted exclusion criteria where we excluded all articles that were extended abstract or not in English. Our process yielded 13 publications. Table I shows a breakdown of selected publications *w.r.t.* each inclusion criteria. Datasets used in our paper is available online [13].

TABLE I
FILTERING OF PUBLICATIONS

Criterion	Count
Initial results (from three databases)	750
Criterion-1 (Puppet research)	33
Criterion-2 (After 2010)	33
Criterion-3 (Non-duplicate)	23
Criterion-4 (Quality)	13
Final	13

III. What quality issues can arise in Puppet scripts?

Similar to production source code, IaC scripts are also prone to issues impacting quality attributes such as security, reliability, and maintainability. Researchers [10], [15], [18], [19] have carried out a detailed exploration about quality issues in IaC scripts. In the remaining section, we elaborate on the key academic explorations associated with IaC script quality.

A. Security smells

Security smells in IaC scripts are recurring code patterns that are indicative of security weaknesses [12]. For example, *hard-coded password* is a security smell in Puppet scripts. In addition to this, Rahman et al. [12] identified six other categories of security smells in Puppet code. We provide the names and definitions of each category in Table II. Experienced practitioners having knowledge in security domain can detect these security smells in IaC scripts through manual inspection. Practitioners may also use static analysis tools, such as SLIC [9] for the purpose.

In addition, Guerriero et al. [3] surveyed practitioners on bad practices related to IaC development concerning Puppet script quality. They also reported *hard-coded values* to be a bad practice while developing IaC scripts. Furthermore, they observed *idempotency* and *documentation* to cause issues in IaC development that were also identified bug categories for IaC. Kumara et al. [6] conducted a grey literature review and identified practices that are detrimental to IaC development. They reported *hard-coded configuration values* and *not using secret management tools* as bad practices.

B. Maintainability smells

Similar to traditional software engineering where code smells are classified based on their granularity as well as scope [17], Sharma et al. [16] classified configuration smells as implementation and design configuration smells. Implementation configuration smells are quality issues such as naming convention, style, formatting, and indentation in IaC scripts. Design configuration smells reveal quality issues in the module design or structure of a configuration project. Table III and IV respectively list implementation and design configuration smells respectively along with their brief description.

In addition, Guerriero et al. [3] identified creating *too large* scripts as a bad practice in their survey of practitioners. Kumara et al. [6] reported *violation of naming conventions* and *insufficient modularization* to be the bad practices for IaC development. Furthermore, Rehearsal [15] detects three types of issues in IaC scripts including *modularity*.

C. Configuration drift and inconsistencies

Configuration drift occurs when a code snippet goes out-of-date because the APIs that it depends on, experience breaking changes over time [5]. Weiss et al. [20] proposed *Tortoise*—a bug repair tool that detects and repairs configuration inconsistencies in Puppet scripts. Tortoise collects a Puppet manifest and user-provided shell commands, and detects inconsistencies between the user-provided commands and the buggy Puppet manifests. Next, Tortoise uses *Z3*, a constraint solver to generate patches that can resolve the detected configuration inconsistencies. Practitioners can find Tortoise handy to detect and resolve configuration drift in Puppet scripts.

Puppet uses configurations specified in manifests and compares that with configurations of the system. This process is referred to as *convergence*. Hannapi et al. [4] used a state transition graph-based approach to detect non-convergent Puppet scripts. The proposed technique can be employed in practice to ensure that the developed manifests are convergent.

D. Bugs and faults

Sotiropoulos et al. [18] proposed a technique to detect dependency-related faults among different types of Puppet resources (such as a service is not tied to its resources). They analyze Puppet scripts and corresponding system call traces and construct a representation containing all the ordering constraints and notifications declared in the program to identify dependency mismatches and potential faults. Bent et al. [19] conducted an empirical study to identify source code elements of a Puppet script that can cause bugs. Similarly, Shambaugh et al. [15] proposed a tool Rehearsal to find three types of bugs in Puppet scripts: *idempotence* and *non-determinism*. Practitioners may find these techniques and tools helpful to locate faults that may occur in their IaC scripts.

IV. How frequently do quality issues occur in Puppet scripts?

To answer this question, we mine open-source repositories that contain Puppet code to measure the frequency of quality issues concerning security and maintainability. The collected repositories were systematically filtered using a selection criteria that included the count of practitioners who developed

TABLE II

BRIEF DESCRIPTION OF DETECTED INSTANCES OF SECURITY SMELLS IN COLLECTED PUPPET SCRIPTS

Security smells	**Description**	**#Instances**
Admin by default	Recurring pattern of specifying default users as administrative users.	45
Empty password	Recurring pattern of using a string of length zero for a password. An empty password is indicative of a weak password.	75
Hard-coded secret	Recurring pattern of revealing sensitive information such as user name and passwords as configurations in Puppet scripts.	6,060
Invalid IP address binding	Recurring pattern of assigning the address 0.0.0.0 for a database server or a cloud service/instance.	175
Suspicious comment	Recurring pattern of putting information in comments about the presence of defects, missing functionality, or weakness of the system.	850
Use of HTTP without TLS	Recurring pattern of using HTTP without the Transport Layer Security (TLS).	681
Use of weak crypto algorithm	Recurring pattern of using weak cryptography algorithms, such as MD5 and SHA-1 for encryption purposes.	94

TABLE III

BRIEF DESCRIPTION OF DETECTED INSTANCES OF IMPLEMENTATION CONFIGURATION SMELLS IN COLLECTED PUPPET SCRIPTS

Configuration smell	**Description**	**#Instances**
Missing Default Case	A default case is missing in a *case* or *selector* statement	179
Inconsistent Naming Convention	Deviates from the recommended naming convention	4
Complex Expression	Contains a difficult to understand complex expression	105
Duplicate Entity	Duplicate hash keys or duplicate parameters	0
Misplaced Attribute	Attribute placement within a resource has not followed a recommended order	67
Improper Alignment	Improper code alignment such as all the arrows in a resource declaration	1,953
Invalid Property Value	An invalid value of a property or attribute is used	51
Incomplete Tasks	The code has "FIXME" and "TODO" tags	459
Deprecated Statement Usage	Deprecated statements (such as "import") are used	219
Improper Quote Usage	Single and double quotes are not used properly	644
Long Statement	The code contains long statements	0
Incomplete Conditional	An "if..elsif" construct used without a terminating "else" clause	182
Unguarded Variable	A variable is not enclosed in braces when being interpolated in a string	158

TABLE IV

BRIEF DESCRIPTION OF DETECTED INSTANCES OF DESIGN CONFIGURATION SMELLS IN COLLECTED PUPPET SCRIPTS

Configuration smell	**Description**	**#Instances**
Multifaceted Abstraction	Elements of the abstraction (e.g. a resource, class, 'define', or module) are not cohesive	3,767
Unnecessary Abstraction	An empty class, 'define', or module	299
Imperative Abstraction	An abstraction contains numerous imperative statements (such as "exec")	99
Missing Abstraction	Resources and language elements are declared without encapsulating them in an abstraction	127
Insufficient Modularization	An abstraction is large or complex	4,699
Duplicate Block	A module contains a duplicate block of statements	807
Broken Hierarchy	Inheritance is used across namespaces where inheritance is not natural	0
Unstructured Module	Ad-hoc structure of a repository	3
Dense Structure	A repository has excessive and dense dependencies without any particular structure	1
Deficient Encapsulation	A node definition or ENC (External Node Classifier) declares a set of global variables	100
Weakened Modularity	A module exhibits high coupling and low cohesion	232

Puppet code, amount of Puppet scripts in a repository, and amount of commits pushed per month. We collect these repositories from three organizations, Mozilla, Openstack, and Wikimedia. We provide attributes of the collected repositories in Table V. We used SLIC [12] and *Puppeteer* [16] to detect and quantify respectively security and configuration smells.

The *Instances* column of Table II shows the count of detected occurrences for each of the security smells. We found that *hard-coded secret* is the most frequent occurring security smell in the analyzed projects. The identified instances reveal sensitive information, such as SSH keys, usernames, and passwords. For example, while analyzing an open-source software repository we found a hard-coded password `v23zj59an`, which was used to login to a database. Revealing such information may pose serious security threats to the organization.

TABLE V

SUMMARY OF THE DATASET

Attribute	**Mozilla**	**Openstack**	**Wikimedia**
Total Repositories	2	61	10
Total Commits	14,449	44,469	71,795
Total Puppet Scripts	1,596	2,845	3,143
Total Puppet-related Commits	6,836	12,227	21,066
Time Period	05/2011-04/2019	09/2011-04/2019	01/2006-04/2019

The second and third most frequent security smells are *suspicious comment* and *use of* HTTP *without* TLS. Though practitioners intend to use `TODO` and `FIXME` to put aspects that they would like to address later, existence of these comments for a sensitive piece of code can provide clues to malicious users to exploit weaknesses. Usage of HTTP without TLS can facilitate 'man in the middle' attacks. The least frequent category is *admin by default*, where user accounts are setup by providing all administrative privileges to a user. Our findings reveal that certain security weaknesses, despite being well-known, occur frequently in Puppet scripts.

Similarly, the 'Instances' column in Table III and Table IV show the number of detected instances for the implementation and design maintainability smells. Among implementation maintainability smells, *improper alignment* occurs the most. A large number of misaligned statements impact the readability, and in turn, understandability of the programs. Two of the smells—*duplicate entity* and *long statement* are not detected at all. From the design perspective, our data suggests that the *insufficient modularization* smell occurs the most indicating that the practitioners put too many resources and their corresponding configuration statements within a single class or module. A related smell is *multifaceted abstraction* that highlights absence of cohesiveness for Puppet abstractions. A high number of *multifaceted abstraction* instances can be linked to *insufficient modularization* smell. It can be inferred that these abstractions are hosting unrelated resource configuration declarations leading to large abstractions. On the other hand, we did not detect any instance of *broken hierarchy* smell and only one instance of *dense structure* smell. *Dense structure* smell is detected at most once per repository and hence, though a small number, is not a surprise.

V. What activities practitioners may adopt to mitigate quality concerns in Puppet scripts?

Recent research [8], [11], [14], [16] has investigated what aspects of IaC scripts and its development are correlated with defective IaC scripts. Rahman el al. [14] identified source code properties that correlate with defects. The study explored defect-related commits along with code changes (commonly referred to as 'diffs') to identify ten source code properties of Puppet scripts. Another study by Rahman et al. [8] identified operations correlating with defects.

To investigate what development activity metrics relate with defective scripts, Rahman et al. [11] conducted quantitative analysis with $2,138$ Puppet scripts collected from 94 open-source repositories. The authors then mine seven development activity metrics; each of these seven metrics maps to a development activity. Sharma et al. [16] proposed a consolidated catalog of 11 design configuration smells and 13 implementation configuration smells. They carried out a detailed empirical assessment of $4,621$ Puppet repositories to observe the characteristics of maintainability smells.

Based on the above-mentioned efforts and their findings, we glean the following observations and recommendations for practitioners to mitigate defects in IaC scripts.

- **Inspect code properties**: Among many code properties supported by configuration frameworks, we advocate practitioners to prioritize inspection and testing efforts for ten code properties (*i.e.,* attribute, command, ensure, file, file mode, hard-coded string, include, lines of code, require, SSH_KEY, complexity, parameter, exec, and lint warnings) [14]. Focusing on these properties can help development teams to reduce the likelihood of defects since these properties are correlated with defects in IaC scripts.
- **Identify and remove security smells**: We recommend detailed assessment and inspection for seven categories of security smells. These smells are listed in Table II with names and corresponding definitions.
- **Identify and remove maintainability smells**: Keeping the source code maintainable is an important aspect to ensure long-term productivity. Findings from Sharma et al. [16] suggest to identify maintainability smells in Puppet code and take appropriate action to refactor them. Table III and IV list the maintainability smells for IaC scripts.
- **Inspect defect-prone operations**: We advocate practitioners to prioritize inspection efforts for scripts that perform any of the below listed operations. These operations have tendency to be correlated with defects and hence a close examination of code performing these operations could help mitigate defects.
  - *File system operations*: File system operations are related to performing file input and output tasks, such as setting permissions of files and directories. While assigning file permissions practitioners can provide the wrong file permissions, *e.g.* assigning `777` instead of `444`. Assigning unrestricted access with `777` can provide unnecessary access to users who do not need such access and is a violation of good access control policies.
  - *Infrastructure provisioning for speciality systems*: This property relates to setting up and managing infrastructure for specialty systems, such as build systems, data analytics systems, database systems, and web server systems. Puppet [7] advertise automated provisioning of infrastructure as one of the major capabilities of IaC tools, but experiments indicate that the capability of provisioning via Puppet scripts can introduce defects.
  - *Managing user accounts*: This property of defective IaC scripts is associated with setting up accounts and user credentials. One of the major tasks of system administrators is to setup and manage user accounts in systems.
- **Avoid development anti-patterns**: Development anti-patterns are recurring development activities that show correlation with defective scripts [11]. We recommend practitioners to look for and avoid the following anti-patterns.
  - *Boss is not around*: The highest contributor usually have the full context of the script, which other practitioners may not have. If a relatively new member of the team contribute more than the highest contributor, the corresponding script have higher chances to be defective. On an average, the highest contributor develops 80%∼90% of the code for non-defective scripts.
  - *Many cooks spoil*: Having multiple practitioners working on the same script increases the defect proneness of

the scripts. Studies [11] reveal that defective scripts are modified by 12∼43 practitioners, whereas, non-defective scripts modified by no more than 11 practitioners.

– *Minors are spoilers*: This anti-pattern states that a script being modified by minor contributors, *i.e.,* practitioner(s) who writes no more than 5% code of the script, makes the script prone to defects. It has been found that a non-defective script may be modified by at most 7 minor contributors, whereas, defective scripts can be modified by 8∼36 minor contributors [11].
– *Silos*: This anti-pattern arises when the practitioners work in disjoint groups. It has been reported that the defective scripts are modified by practitioner groups, which are 1.3∼2.0 times more disjoint, on average, compared to that of non-defective scripts [11].
– *Unfocused contribution:* The anti-pattern occurs when a practitioner working on an IaC script also modifies other scripts. On an average, unfocused contribution for defective scripts accounts to up to 95% higher than non-defective scripts [11].

VI. Challenges and Opportunities

Despite advances in the domain of IaC script quality, there exists research gaps that need to be filled to address challenges faced by practitioners:

- *Lack of assessment and automated refactoring tools*: Traditional software engineering enjoy the benefits of the availability of comprehensive tool support for detecting various quality issues and, to some extent, refactor them automatically. However, currently, IaC domain lacks similar extensive tool support, which makes the task of developing good quality IaC scripts challenging.
- *Poor actionability and integration*: Ad-hoc static analysis tools for various kinds of defects can at most scratch the surface of the real issues. It is necessary that the developed tools are integrated with development environments to highlight the issues earlier. In addition, a tighter integration reduces the efforts from the development side to explicitly carry out assessments for code quality.

VII. Conclusion

In continuous deployment, IT organizations rapidly deploy software and services to end-users using an automated deployment pipeline. IaC is a fundamental pillar to implement an automated deployment pipeline. Defective IaC scripts, such as defective Puppet scripts can hinder the reliability of the automated deployment pipeline. By mining Puppet scripts from the OSS domain we have synthesized a set of lessons that will help practitioners to improve the quality of Puppet scripts. In particular, we list security and maintainability smells that may occur in configuration code. We used existing tools to measure the frequency of those smells in Puppet repositories of three well-known open-source organizations. Finally, we discuss challenges and opportunities in the domain of IaC for both the industry and academic communities.

Acknowledgements

We thank the PASER group at Tennessee Technological University (TnTech) for their valuable feedback. This research was partially funded by the U.S. National Science Foundation (NSF) Award # 2026869.

References

[1] V. Basili, L. Briand, D. Bianculli, S. Nejati, F. Pastore, and M. Sabetzadeh, “Software engineering research and industry: a symbiotic relationship to foster impact,” *IEEE Software*, vol. 35, no. 5, pp. 44–49, 2018.

[2] L. Briand, D. Bianculli, S. Nejati, F. Pastore, and M. Sabetzadeh, “The case for context-driven software engineering research: Generalizability is overrated,” *IEEE Software*, vol. 34, no. 5, pp. 72–75, 2017.

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

[4] O. Hanappi, W. Hummer, and S. Dustdar, “Asserting reliable convergence for configuration management scripts,” *SIGPLAN Not.*, vol. 51, no. 10, pp. 328–343, Oct. 2016.

[5] E. Horton and C. Parnin, “V2: Fast detection of configuration drift in python,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 477–488.

[6] I. Kumara, M. Garriga, A. U. Romeu, D. Di Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel, “The do’s and don’ts of infrastructure code: A systematic gray literature review,” *Information and Software Technology*, vol. 137, p. 106593, 2021.

[7] Puppet, “Continuous delivery,” https://puppet.com/products/capabilities/automated-provisioning, 2021, [Online; accessed 14-Nov-2021].

[8] A. Rahman and L. Williams, “Characterizing defective configuration scripts used for continuous deployment,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, April 2018, pp. 34–45.

[9] A. Rahman, “Security Linter for Infrastructure as Code Scripts (SLIC),” 05 2019. [Online]. Available: https://github.com/akondrahman/IacSec

[10] A. Rahman, E. Farhana, C. Parnin, and L. Williams, “Gang of eight: A defect taxonomy for infrastructure as code scripts,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20. Association for Computing Machinery, 2020, p. 752–764.

[11] A. Rahman, E. Farhana, and L. Williams, “The ‘as code’ activities: Development anti-patterns for infrastructure as code,” *Empirical Softw. Engg.*, no. 25, 2020.

[12] A. Rahman, C. Parnin, and L. Williams, “The seven sins: Security smells in infrastructure as code scripts,” in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE ’19, 2019, pp. 164–175.

[13] A. Rahman, M. Rahman, C. Parnin, and L. Williams, “Dataset for Security Smells for Ansible and Chef Scripts Used in DevOps,” 7 2019. [Online]. Available: https://figshare.com/s/9f6f1c5bfa6cca9b9214

[14] A. Rahman and L. Williams, “Source code properties of defective infrastructure as code scripts,” *Information and Software Technology*, 2019.

[15] R. Shambaugh, A. Weiss, and A. Guha, “Rehearsal: A configuration verification tool for puppet,” *SIGPLAN Not.*, vol. 51, no. 6, pp. 416–430, Jun. 2016.

[16] T. Sharma, M. Fragkoulis, and D. Spinellis, “Does your configuration code smell?” in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR ’16, 2016, pp. 189–200.

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

[18] T. Sotiropoulos, D. Mitropoulos, and D. Spinellis, “Practical fault detection in puppet programs,” in *Proc. of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, p. 26–37.

[19] E. van der Bent, J. Hage, J. Visser, and G. Gousios, “How good is your puppet? an empirically defined and validated quality model for puppet,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 164–174.

[20] A. Weiss, A. Guha, and Y. Brun, “Tortoise: Interactive system configuration repair,” in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017, 2017, pp. 625–636.