Code Smells in Infrastructure as Code

Julian Schwarz
RWTH Aachen University
Aachen, Germany
julian.schwarz@rwth-aachen.de

Andreas Steffens
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
steffens@swc.rwth-aachen.de

Horst Lichter
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
lichter@swc.rwth-aachen.de

***Abstract*—Ensuring high quality in software systems is a well-known and big challenge. Infrastructure as Code (IaC) gathered increasing popularity in recent years, but there is only little research done in terms of quality of this code. Like with programming languages we find a high diversity of languages and technologies. Existing research introduced code smells from traditional software engineering to the popular provisioning tool Puppet, which uses IaC to specify the desired state of environments. Results show that code smells are an adequate method to assess the quality of Puppet code.**

In this paper we extend the existing research by first applying code the IaC smells to an other technology and investigate if similar results can be achieved. We applied the code smells in two case studies to open and closed source IaC code repositories. The presented results indicate that IaC smells are present in other tools and technologies. Furthermore the results show that IaC smells are agnostic to the applied technology and can be defined on a technology agnostic level. Second, we introduce new code smells from the field of software engineering, which were not covered yet, to the domain of IaC. The paper presents a catalogue of 17 code smells which were applied to Chef and whose implementation is available as Open Source.

I. Introduction

Infrastructure As Code is an emerging practice to describe and specify the underlying infrastructure of software systems and their configuration. Within a DevOps-oriented development process IaC helps to automatically build, provision, configure and manage a software system with each change of the software. IaC covers the specification of virtual hardware using tools like Terraform [1]. Other tools like Puppet[2] and Chef[3] define the required installed software products and their configuration as IaC. As IaC evolves over time it is a good practice to store IaC files alongside with the source code of the respective software.

DevOps emphasizes to apply methods and practices from development to operation and vice versa. Code smells are a well known for programs, so applying the idea of code smells to IaC is obvious. But to the best of our knowledge there is only very little research available. This paper extends the work of Sharma et al. [1] who introduced IaC smells for the first time focusing on Puppet and finally present a tool to detect them in Puppet files. Puppet is a configuration management tool which defines an own declarative language to describe infrastructure. Chef is another often used configuration management tool offering a Ruby-based declaration language. Though there are documented best practices for Chef and especially for Ruby as well as style guides, a comprehensive list of code smells for Chef is still missing. Obviously, there is neither tool support to detect Chef IaC smells nor a refactoring approach to remove those smells systematically.

In this paper we want to answer the following questions;

- Are the Puppet IaC smells applicable to Chef or other tools and technologies or even to complete domain of IaC?
- If yes, do we obtain the similar results?
- Are there additional relevant IaC smells not published yet?

Therefore, at first we transfer a selection of Puppet IaC smells to Chef IaC to validate the results of Sharma et al. and to show that the these IaC code smells definitions are general and can be applied to different technologies and tools. Furthermore, we propose a set of new IaC smells based on well-known smells published in books like Fowler's Refactoring book [2]. Our overall aim is to propose an initial catalog of IaC smells for Chef, which can be further extended to improve the quality of Chef cookbooks.

This paper is organized as follows. At first, we give some background information on configuration management and code smells in Section II. Thereafter, in Section IV, we discuss our research questions followed by the presentation of our solutions and the results we obtained performing a case study (Section V). We conclude this paper by summarizing our results in Section VIII.

II. Background

A. Code Smells

The concept of code smells was introduced by Kent Beck and Martin Fowler [2] and describes flaws in code which may lead to problems. A code smell does not lead to a run-time error but usually indicates that the code needs to be improved. An well-known example is the Duplicate Block (DB) smell, which occurs if a sequence of identical statements is repeated throughout the code and should be moved to a method.

B. Infrastructure as Code

Configuration management eventually was born out of the need to maintain systems with different architectures,

[1]www.terraform.io
[2]www.puppet.com
[3]www.chef.io

software versions and configuration settings inside the same infrastructure [3]. In 1993, Mark Burgess developed the first configuration engine CFEngine [3] to deal with different architectures and operating systems without the need to maintain innumerable configuration scripts. IaC is directly linked to configuration management and is basically a synonym. Configuration management tools like Chef or Puppet translate infrastructure code into infrastructure [4].

Puppet's configuration code is a type of IaC. Resources are the smallest configuration components. Multiple resources can be consolidated in classes and multiple classes, including other files needed, can be joined in a module. A module configures a part of an infrastructure like e.g., a mail server.

Chef, another configuration management tool, defines the following three major components:

- Resource: smallest unit, specifies one functionality
- Recipe: consists of one or more resources and installs a component of an infrastructure part
- Cookbook: consists of one or more recipes and describes an infrastructure or one of its parts

Chef recipes are written in a domain-specific language, based on Ruby, in a declarative way. In comparison to Puppet however, it is possible to specify configurations in an imperative way, as cookbooks are written in Ruby. A Chef client is a binary file which interprets the cookbook(s). It installs the infrastructure according to the cookbook.

III. Related Work

Current research regarding IaC focuses on specific quality attributes of infrastructure code, idempotency and convergence. Hummer et al. [5] applied techniques from model-driven testing to Chef and were able to find non-idempotent cookbooks. Shambaugh et al. introduced Rehearsal [6], a configuration verification tool for Puppet, using static analysis. Rehearsal can verify two central qualities of Puppet manifests, determinism and idempotency. Wettinger [7] proposed an alternative to the convergence oriented approaches using concepts called compensation and virtualization. Theses approaches assess very specific properties of IaC but fail to provide information to developers where and how to improve the infrastructure code.

More recent research was conducted to assess more quality attributes of IaC. Bent et al. [8] derived a quality model for Puppet based on an empirical study, while Rahman and Williams [9] used text mining techniques. Cito et al. [10] extended the scope to Docker while Jha et al. [11] took a look on Android manifests. All these approaches adapt existing ones, which has been proven successful in other areas of software engineering like defect prediction or testing.

Sharma et al. [1] proposed to apply the code smell metaphor for IaC as well. Fowler [2] claims that code smells have a negative impact on the quality of code and are a main indicator to refactor the code. In their study [12], Olbrich et al. conclude that *classes infected with code smells have a higher change frequency, meaning that such classes seem to need more maintenance than non-infected classes*. Fontana et al. analyzed in their study [13] the frequency of code smells in systems of different domains. The analysis yielded that all domains except of one incorporate a common set of code smells.

Sharma et al. [1] applied the principles of code smells to Puppet. They propose a catalog of 24 smells. They distinguish between implementation and design smells. While implementation smell involves quality issues like naming conventions, style, formatting and indentation, design smells involve quality issues in the *module design or structure of a configuration project*. The authors applied their smells on public available configuration code. We refer to these specific smells as Puppet smells in the rest of this paper. In our study we compare our findings with these previous results.

IV. Methodology

Based on our initial motivation as stated in Section I and to define the scope of this work, we formulate the following two research questions.

- **RQ1** *Are Puppet smells applicable to another configuration management tool like Chef and can we conclude that they are applicable to the complete domain of IaC?*
  To answer this question, we pick the five most frequently occurring implementation configuration smells and the five most frequently occurring design configuration smells from the smell catalog proposed by Sharma et al. [1] and convert them into detection rules for Foodcritic, a static code analysis tool designed for Chef. To achieve this we analyzed the code smells and classify them into three categories. Each category indicates the necessary actions to apply a smell to specific technology. To validate our approach, we apply the Chef smells to public available Chef cookbooks. By comparing our results with the results of Sharma et al. [1] we are able to answer our research question.
- **RQ2** *Are there additional and well-known programming smells which are relevant for IaC?*
  Following our previously used practice, we first create abstract smell definitions based on published code smells definitions and then define the respective concrete IaC versions. There, we focus on smells from Fowler's Refactoring [14] book. Furthermore, we preferred more complex smells, respectively design and architecture smells, which are not contained in the Puppet smell catalog. In addition, we redefined violations against best practices for Chef as smells. All these new smells were implemented using Foodcritic and applied to the same cookbooks as before.

V. Results

A. Abstract Smell Classification

As many of the Puppet smells and their detection methods are specific to Puppet, we were not able to transfer certain Puppet smells directly to Chef.

TABLE I
SMELL MATRIX

	Technology Agnostic	Technology Dependent	Technology Specific
Implementation	Improper Alignment, Long Statement, Unguarded Variable, Misplaced Attribute, *Long Resource*	Improper Quote Usage	Hyphens
Design	Multifaceted Abstraction, Duplicate Block, *Avoid Comments*, *Too many Attributes*	Insufficient Modularization, Weakened Modularity, Unstructured Module, *Law of Demeter*, *Include Consistency*	Empty Default

We discovered that there are at least three kinds of IaC smells: technology agnostic smells, technology dependent smells and technology specific smells.

- **Technology Agnostic Smells** These smells can adapt from the respective Puppet smells without changing the defined detection method (often based on metrics, see Marinescu [15]). Obviously, we may need to change the implementation of the smell detection method as we need to consider the different syntax of the languages. An example of a technology agnostic smell is Duplicate Block. It can be detected for blocks with a specific count of characters no matter which characters and thus it can be detected in any language, even in natural language.
- **Technology Dependent Smells** These are Puppet smells that are not directly applicable to Chef IaC due to various differences between Puppet and Chef, e.g., comprise of concepts that exist for Puppet but not for Chef.
  To apply a technology dependent smells to Chef, we need to alter its detection method. Therefor, we identified the needed smell detection information and the structures in Chef, which provide these information. We used both to define the corresponding Chef specific detection methods for those smells. To understand the structures and concepts of Chef we primarily consulted the official Chef style guide [16], an additional community style guide [17] and the Chef documentation [18].

Listing 1. Technology Dependent Smell - Modules in Puppet

```
# Download Java
remote_file java do
        source url
        owner 'user'
        group 'group'
        action :create_if_missing
end
```

Listing 2. Technology Dependent Smell - Modules in Puppet

```
class java::install {
        file { 'java':
                source => url,
                owner => 'user',
                group => 'group,
                ensure => present
        }
}
```

  An example for a technology dependent smell is the Insufficient Modularization smell. Puppet uses a concept called class to structure configuration code, in Chef a similar concept is called recipe. While the class concept of Puppet is realized by the keyword c̈lass,̈ Chef's DSL does not provide such a keyword. Instead, recipes are distinguished by using different files. Listing 2 shows a Puppet module inside a class structure, the equivalent structure for Chef in shown in Listing 1. Due to the different realizations of modules, the detection methods of technology dependent smells need to by changed.
- **Technology Specific Smells** These kinds of smells can be only applied to one specific IaC technology. We derive these smells from common best practices and style guides. The absence of these practices indicate a code smell. For Chef we identified two technology specific smells. One is called Empty Default smell which claims that the "default.rb" recipe should always be empty. This is very specific to Chef, since there is no such concept of recipes in Puppet.

Based on our analysis, we created abstract smell definitions which should be applicable to every IaC. Moreover, we created a classification, especially the distinction between technology agnostic and technology dependent smells. Based on these results we created concrete smell definitions for Chef. We suggest this as a process of transferring smells in the domain of Infrastructure as Code (IaC).

Note, that the abstraction of a smell also provides an abstraction of the its detection method (and the used metrics) for technology agnostic smells and therefore simplifies its application. The concrete implementation of the detection method needs be adapted in all cases.

B. Catalogue

Finally, we defined a catalog of Chef IaC smells distinguishing between , technology agnostic and technology dependentThe definition of all smells are shown in Table II, Table III and Table IV. Table Table I shows in overview of our classification with respect to the distinction between implementation and design smells. It's obvious that most of the implementation smells are technology agnostic due to their nature of being rather simple. They cover mostly style issues. For the more complex design smells its harder to find a technology agnostic definition as they need more sources of information and the metrics are more complex. But the results

TABLE II
TECHNOLOGY AGNOSTIC SMELLS

Smell	Description	Detection
Improper Alignment (IA)	The smell is detected if there are any tabulation characters, or if the Resources parameters are not correctly and consistently indented.	Tabulator characters are detected, by scanning each line of a file whether it contains the string "$\backslash t$". For the detection of the proper alignment, we identify the location of the first non whitespace character for each inside a block.
Long Statement (LS)	This smell describes code lines or statements which are too long and thus typically do not fit in a screen.	The smell is detected by counting the number of character of each line. The limit is 140.
Unguarded Variable (UV)	The UV code smell wants to make sure that variables in strings are properly interpolated. Puppet interpolates a variable via ${}, while Chef interpolates variables like this #{}. The smell occurs if a variable is not enclosed in braces when being interpolated in a string.	We use the AST to collect all variables. Then we extract via RegEx all strings in quotes (single and double quotes) and search in those strings for a # following a variable.
Misplaced Attribute (MAt)	According to the official Chef styleguide [16] the specific order of information inside a resource should be followed.	We identify resources in the AST. For each resource we fetch the used parameters. Each parameter has a predefined priority number based on style guides. From this we compute the smell by comparing the list of parameters the ordered list of possible parameters.
Multifaceted Abstraction (MA)	MA for Chef means that each recipe and each resource should only specify the properties of a single piece of software.	We suggest to scan *execute* resources for commands using concatenation or piping like "&&", "|" or ";".
Duplicate Block (DB)	This smell describes blocks of statements occurring more than once	If a code block of at least 150 characters occurs more than once, the smell is present. Literature provides clone detection algorithms, e.g. using AST [19]. We use a simple search-based algorithm.
Long Resource (LR)	This smell is mapped from the Long Method smell by Fowler [2]. As for Chef all resources, except bash and execute resources, are limited in length by the number of their parameters and the corresponding values. The same takes effect for bash and execute resources, besides the value of the command attribute in execute resources is not limited and code in bash resources is not limited, too. Therefor, we mapped this smell to be applied only to bash and execute Resources and changed the name to *Long Resource* instead of *Long Method*. Execute and bash Resources being too long should be detected.	We calculate one metric, the number of LoC inside bash and execute Resources. We detect the smell if those Resources have more than 7 LoC. We count the LoC, by traversing the AST and counting how many different line numbers the Resource comprises.
Too many Attributes (TmA)	The TmA smell is derived from the Long Parameter List smell combined with the Speculative Generality smell by Fowler [2]. We combine the ideas of both of the smells and recommend to not use attributes excessively to simplify code and minimize the effort needed for maintaining the code.	As a metric, we calculate the number of variables per lines of code. If the calculated value is greater than 0,5 the smells is detected. We suggest to search the AST for variables and variable references. Those are saved in an array and the size of the array is finally divided through the number of lines.
Avoid Comments (AC)	Based on Fowler [2] comments often lead to bad code or are used to as a deodorant for bad code. Additionally Robert C. Martin deals in detail with comments being a possible code smell [20] by naming a few cases like redundant comments, bad comments or commented code. Following the suggestion of Fowler and taking Robert C. Martins considerations into account we advise not to use comments at all. Since Chef is declarative and has comprehensive resource names, normal resources are always self-explaining. Comprehension problems might occur on *bash* or *execute*, custom resources, etc.. In those cases the resource shall either follow single responsibility principle or needs to be splitted up.	For Chef we decided to detect the smell on every comment, except all comments until the first non-commented line. We exclude the first comments, because they often include licensing information.

indicate that design smells are not technology specific which supports our assumption for our first research question, that code smells are a applicable to the complete domain of IaC.

The implementation of the detection methods for all Chef smells was done in Foodcritic and can be found as open source under the following URL [22]: https://github.com/swc-rwth/InfrastructureAsCodeSmells

For answer the second research question, we identified five new code smells applicable to the IaC. The field of software engineering provides a extensive collection of code smells and anti-patterns [2], [23]. The newly identified smells are included in our catalogue and written in italic.

C. Completeness & Soundness

Soundness and completeness, as introduced by Jalote [24]: are important properties of a static code analysis approach. While soundness captures the occurrence of false positives in the discovered defects, completeness characterizes how many of the existing defects are not discovered by static code analysis. As full soundness and completeness is not possible, a static code analysis should be as sound and as complete as possible.

The authors claim that there is a trade off involved. A higher degree of completeness often implies less soundness, that means more false positives.

TABLE III
TECHNOLOGY DEPENDENT SMELLS

Smell	Description	Detection
Improper Quote Usage (IQU)	For Chef we demand not to quote Booleans, not to use variables in single quoted strings and to quote resource titles, with the exception of variables as resource titles.	We detect the smell using string search combined with regexes.
Insufficient Modularization (IM)	The IM smell is defined as to avoid abstractions which are too large or complex. For Puppet [1], the smell includes three cases which indicate an abstraction which is too large or complex. 1) a file with more than one classes 2) class declaration too large 3) class declaration too complex We can only adapt the last two cases for Chef by replacing *class* by *recipe* and thus defining the second case as *Recipes being too large* and the third case as *Recipe complexity too high*.	To detect the properties of the smell, so Recipes being too large and Recipes being to complex, we use the following two metrics. • the number of lines should be below 40 • the maximum nesting depth: each *do_block* and *if* in the AST increase the nesting depth and therefor complexity To compute the nesting depth we scan the AST for the nesting elements *do_block* and *if*.
Weakened Modularity (WM)	For Chef we demand Cookbooks, Recipes and Resources to follow the high cohesion, low coupling principle.	The coupling is determined by the two metrics number of *includes* which refers to *import coupling* [14] and the cohesion by the number of LoC without *includes*. We therefor calculate the ratio of the number of includes of other cookbooks and the LoC and detect the WM smell if this ratio is greater than 0.1.
Unstructured Module (UM)	Based on this [17] Chef styleguide we define the following metrics for the UM smell: • absence of the required *metadata.rb* file • absence of the recipe folder indicating a poorly structured cookbook (attributes all specified in recipes), as well as • absence of the attribute folder • *.rb* file ending for all files in the recipe folder • *.rb* file ending for all files in the attribute folder • *.erb* file ending for all files in the template folder	We scan the directories for each cookbook for the missing metadata file and missing attributes / recipes folders. For the file endings we just pull all files of the respective folders and check their file endings.
Law of Demeter (LoD)	The LoD is a well known design guideline discovered by Ian Holland, originally formulated for object-oriented systems, and put down by Professor Karl Lieberherr with the following general definition [21]: *Each unit should have only limited knowledge about other units: only units "closely" related to the current unit. Or: Each unit should only talk to its friends; Don't talk to strangers*. For Java this means e.g. that a class A should not invoke methods on a class C by calling a getter of a class B which returns a reference to that C class (which belongs to B). This is also described more narrow in the object oriented defintion of the LoD. For Chef we want to avoid transitive dependencies. That means, if one Chef cookbook A includes another Chef cookbook B, but both of them include a cookbook C, the smell is discovered.	All cookbooks (so all recipes of all cookbooks) which are included in the respective cookbook are examined whether they also include one of the dependencies which are included in the respective cookbook. Considering implementation we require all cookbooks and their dependencies and therefor all dependencies of all cookbooks to be in the same directory. The first step is to scan all recipes of the respective cookbook for includes and store them in an array. In doing so, we exclude includes of recipes of the same cookbook, distinguishable by strings with the cookbook name of the respective cookbook followed by 2 colons. The second step is to scan all other cookbooks if they also include cookbooks out of the array. If so, the smell is detected. The third step is to scan all recipes (if the respective cookbook included only a recipe of a cookbook.
Include Consistency (IC)	This smell is adapted from the LoD smell and they are closely related. Where the LoD searches for exact matches of transitive includes in other cookbooks, the IC smell searches for transitive includes of cookbooks where the cookbook name suggests a similar functionality.	The detection strategy is very similar to the one of the LoD, except for not looking for exact matches.

TABLE IV
TECHNOLOGY SPECIFIC SMELLS

Smell	Description	Detection
Hyphens	According to to the Chef styleguide [17] hyphens in cookbook names should be avoided.	We just scan the folder of the cookbook for hyphens.
Empty Default (ED)	According to the Chef styleguide [17] one should create a default.rb, but it should be empty and should not include any *includes*. Instead there should be other Recipes with specific functionalities and the user should know about them.	The smell occurs if either there is no default.rb or it is empty. Therefor, we just scan for the existence of a default.rb and check if the file is empty by checking if the parser outputs nil.

As in our case a true positive does not discover a defect, but indicates the existence of a smell, we focused on the soundness and therefore accept a lower degree of completeness. To assess and improve the soundness of each smell detection method, we conducted the following process: First, we applied the smell detection method on at least 10 sample occurrences, preferably in 10 different cookbooks. If the detection method produced too many false positives we iteratively improved the implementation or even modified the used metrics of the detection method until no more false positives were found in the samples. The iterative evaluation and improvement of the smell detection method regarding completeness and soundness is a necessary step to ensure acceptable detection results. For example, when developing the detection method for the Long Resource (Long Statement) smell, we initially limited the scope of the detection on a defined set of resources to ensure a high level on soundness. By changing the scope to all resources the detection quality was improved drastically.

VI. Evaluation

In order to evaluate our approach and our results, we conducted two case studies and examined the property of soundness and completness.

The main objective of these case studies was to validate the proposed Chef smells and to compare their application with the previously published results for Puppet smells. The case studies were based on two Chef cookbook datasets. Both cookbook datasets, one representing the official Chef supermarket and the other one a cookbook repository from industry, were analyzed using the same code smell detection methods. We executed the smell detection methods using Foodcritic 11.3.0 and Ruby 2.4.0. All developed scripts can be also found on GitHub [22].

A. Completeness & Soundness

Soundness and completeness, as introduced by Jalote [24]: are important properties of a static code analysis approach. While soundness captures the occurrence of false positives in the discovered defects, completeness characterizes how many of the existing defects are not discovered by static code analysis. As full soundness and completeness is not possible, a static code analysis should be as sound and as complete as possible.

The authors claim that there is a trade off involved. A higher degree of completeness often implies less soundness, that means more false positives.

As in our case a true positive does not discover a defect, but indicates the existence of a smell, we focused on the soundness and therefore accept a lower degree of completeness. To assess and improve the soundness of each smell detection method, we conducted the following process: First, we applied the smell detection method on at least 10 sample occurrences, preferably in 10 different cookbooks. If the detection method produced too many false positives we iteratively improved the implementation or even modified the used metrics of the detection method until no more false positives were found in the samples. The iterative evaluation and improvement of the smell detection method regarding completeness and soundness is a necessary step to ensure acceptable detection results. For example, when developing the detection method for the Long Resource (Long Statement) smell, we initially limited the scope of the detection on a defined set of resources to ensure a high level on soundness. By changing the scope to all resources the detection quality was improved drastically.

B. Case Study: Industry Repository

We analyzed an internal Chef supermarket of the our industry partner on the occurrence of smells. Thereby, we scanned 35 cookbooks with 293 files including 184 recipes. Those Recipes have a total of 8379 Non-Blank Lines of Code (NBLoC). Figure 1 shows the total count of occurrences per smell which sum up to 477 detections.

We can see four rules hitting over 40 occurrences. Those are 62 usages of tabulator characters and 55 indentation flaws, 49 occurrences of IM, 62 occurrences of duplicate block and 86 Recipes with comments.

Important though is, that those numbers do not directly reflect the quality of the code and that the number itself does not necessarily reflect the representation of the smell. We used this repository to investigate our smell definitions and implementation in deep. During this we identified a lot of flaws, e.g. the first definition of smells making use of chef's modularization concepts like recipes did not evaluate extensions made to Chef using custom resources, templates and libraries. We repeated our study with adapted and extended smell metrics. The results are shown in Figure 2. The total number of smells is 648. There is an increase for the smells IQU, LS, IM, DB and AC. Those are mostly smells which can be detected in any text file (IQU,LS,DB), which among others leads to many false positives in templates, or any ruby file (AC). Concerning the IM smell, there is an increase from 49 to 89 detected smells.

Further analysis shows, that all of the 9 additional occurrences of the nesting depth rule are because of custom Resource definitions, which is immense, because those 9 occurrences only emerge out of 23 custom Resource files, whereas we analyzed 184 Recipes.

About the half of the additional occurrences of LR are also detected for custom resources, the other half for templates. All in all this analysis shows, that especially the custom resources need refactoring in terms of complexity and maintainability. We can conclude that the analysis of cookbooks using the extended smells provides insights about custom resource definitions. Although the risk of false positives is increased.

C. Case Study: Supermarket

In the context of this case study we analyzed about 3200 cookbooks out of the official supermarket. We first analyzed the total occurrences per smell as shown in Figure 3. 44230 smells were detected on 372254 NBLoC. The first conspicuousness is the high number of 23676 (capped at 5000 in Figure 3) occurrences of the IA smell, where 22157 occurrences

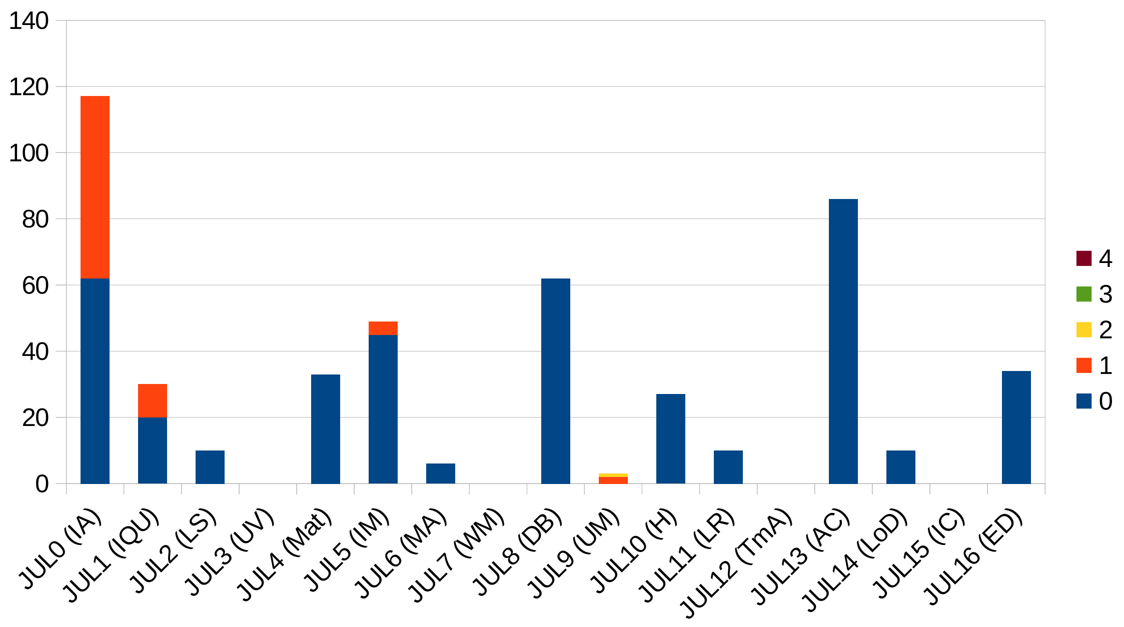

Fig. 1. Total occurrences of IaC smells in industrial repository

are solely caused by the tabulator character symptom of the smell. To have a better scaling, we thus limited the x-axis to 5000 in Figure 3.

Apart from the IA smell, there are more smells like AC, ED and LS which occur more than 2500 times in the dataset.

Regarding our first research question, we see that nearly each of the proposed smells was detected. Only the UV smell was not detected. Its interesting to see that technology agnostic design smells like IM and UM occur very often and more often than many implementation smells. This indicates that the current way of developing IaC already benefits from existing support in IDEs and editors. But for detecting technology-dependent design smells more effort has to be done. In general the findings are quite comparable to the industry repository. Therefore, we conclude that code smells detection can be applied to IaC in general to assess quality. The second research question we see, that the new introduced smells can be identified in a relevant degree.

D. Comparison Puppet Smells

Sharma et al. [1] decided to download all Puppet repositories with at least 40 commits to ensure a certain maturity of the puppet code. All of them together comprise 132662 Puppet files and 8948611 LoC.

Thereby, they distinguished the total number of detected smell instances by volume and existence. Volume thereby means the total number of occurred smells within a project, whereas existence describes the number of how many different types of smells were detected within a project.

Table V shows the volumes of the top smells in chef and their corresponding result for Puppet. Position one to three reflect the top three implementation configuration smells and position four, five and six the top three design configuration smells. Taking only the technology agnostic and technology dependent smells into account, results of Puppet and Chef are nearly identical, e.g. same top smells, IA and LS. Interesting are the differences between Puppet and Chef. The IQU smell is one of the top three smells in Puppet but in Chef it can be found on position seven. We could not identify a direct reason but assume that the DSL of Chef is more likely to prevent this smell, e.g. quoting is more mandatory in Puppet. Even more interesting the the absence of the UV smell in Chef, where as in Puppet it ranks quite high. For MA smell the difference occurs, because our implementation only considers a small set of resources and do not consider inter-resource resource cohesion. In this case the more strict DSL from Puppet eases the detection of this kind of smells.

If we compute the frequency of the smells in relation to the

TABLE V
TOP 10 CHEF SMELLS VS. PUPPET

Smell		Chef		Puppet		Class
Rank	Smell	Occ.	Freq.	Occ.	Freq	
1	IA	23,676	15	780,265	11	I/TA
2	LS	2,823	131	527,637	16	I/TA
3	MAt	2,336	159	22,976	389	I/TA
4	IM	2,075	179	96,033	93	D/TD
5	UM	1,806	206	4,653	1923	D/TD
6	DB	1,201	309	17,601	508	D/TA
7	IQU	1,108	335	428,951	20	I/TD
8	MA	449	829	64,266	138	D/TA
9	WM	29	12836	13,944	641	D/TD
10	UV	0	-	71,339	125	I/TA

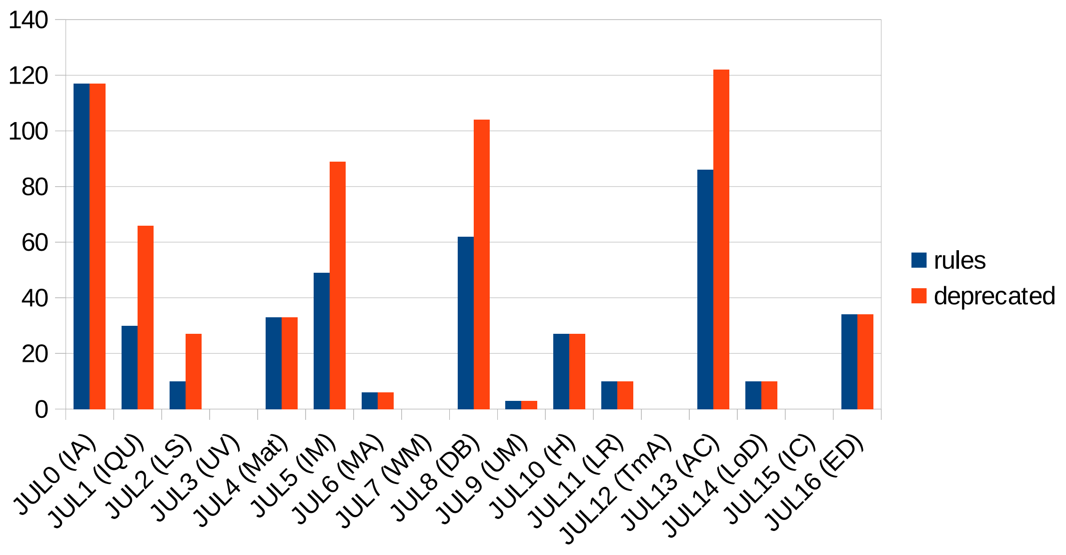

Fig. 2. IaC adapted code smells for industry repository

amount of code , we can identify that only a few smells are more likely to occur in Chef than in Puppet. This might be due to the focus on sound rules and thus not having complete rules, which means we do not detect each occurrence of a smell or the Chef supermarket has a higher code quality. Another aspect may be, that the used metrics and the used smells are not suitable to evaluate the quality of Chef cookbooks at all. There may be better metrics which detect quality issues in Chef code more accurately.

The comparison shows, that code smells occur in both technologies. We find a similar rankings of smells, which leads to the assumption that IaC suffer from similar problems and developing a catalogue of general IaC code smells is needed. Further more the comparison shows, that the smell detection is highly sensitive to defined metrics and the choosen implementation. So far it is not advisable to compare IaC from different technologies to identify the best language or tool.

VII. Threats to Validity

As a threat to validity we regard our choice of metrics for Chef. As seen in the evaluation, broadening the scope of metrics, respectively changing them has a big impact on the results and thus is a big factor to be considered. Furthermore, the implementation of the detection contains a certain risk, since e.g. deciding whether a rule is rather sound than complete gives varying results regarding false negatives and false positives as stated in Section VI-A. During our study we encountered several bugs in Foodcritic which may influence the detection of smells, e.g. the erroneous detection of recipes breaks the defined detection scope. Other implementation issues lead to rules not being sound as wanted.

VIII. Conclusion and Future Work

The aim of this paper was to identify code smells in IaC, more specifically we focused on Chef. Based on the foundation done by Sharma et al. [1] in their paper "Does Your Configuration Code Smell", we formulate two research questions. For the first research question, we adapted the top 10 Puppet smells for Chef. Thereby, we introduced a classification to be able to distinguish between technology agnostic and technology dependent smells. The comparison with Sharma et al. showed an accordance of detected smells and of a similar distribution in the data. Therefore, we conclude, that these smells are adequate to be used to investigate the quality of IaC in general. We identified more code smells from the field of software engineering, which can be applied to IaC. The application to Chef code indicates a high relevance of these smells. In addition our newly defined smells cover more complex design related problems and offer therefore more valuable results. Due to their nature of being technology dependent smells, design smells are more complex to implement and more sensitive to the specific choice of the metric used to detect the smell. Our paper shows that the previous results are valid and transferable to other kinds of IaC. This indicates that a further investigation of code smells in the area of Infrastructure As Code is needed. We identified several aspects for future research. e.g. other configuration and provisioning tools using IaC like Ansible, Salt and Terraform needs to be investigated to validate our results. Our catalogue of IaC smells is not complete. Further existing code smells from software engineering can be investigated, especially design and architectural smells needs to be taken into account. Based on code smells known refactoring practices can also be applied to IaC. In software engineering research code smells are often used to calculate risk or technical of a software project.

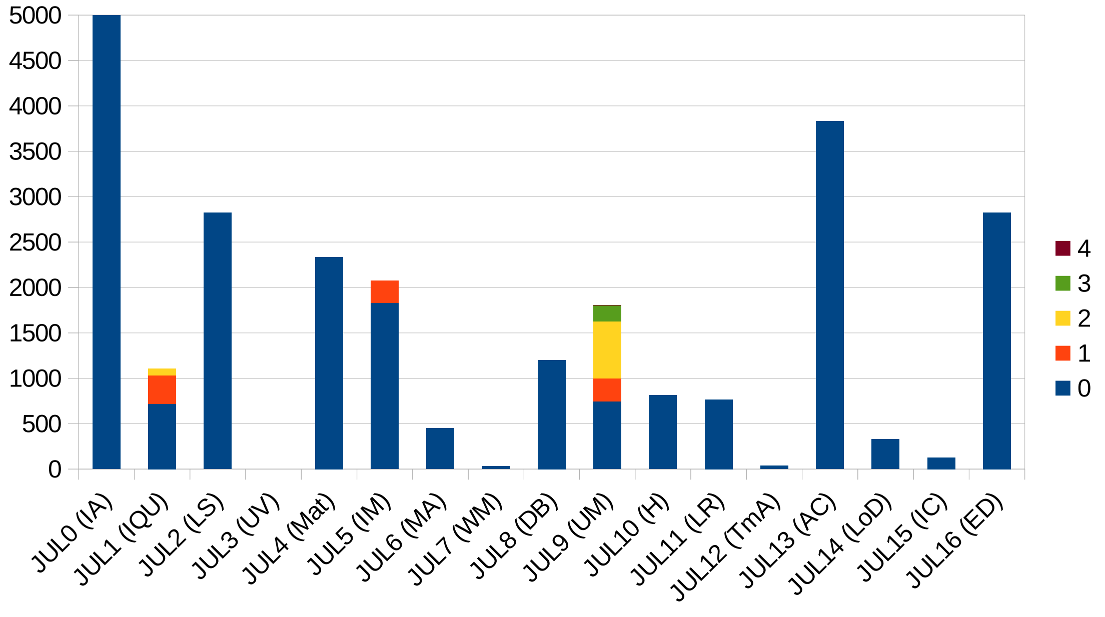

Fig. 3. Total occurrence of IaC smells in public Chef supermarket

References

[1] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 189–200.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] J. Cowie, *Customizing Chef: Getting the Most Out of Your Infrastructure Automation*, 1st ed. O'Reilly Media, 9 2014. [Online]. Available: http://amazon.com/o/ASIN/149194935X/

[4] K. Morris, *Infrastructure As Code: Managing Servers in the Cloud*. Oreilly & Associates Incorporated, 2016. [Online]. Available: https://books.google.de/books?id=kOnurQEACAAJ

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

[6] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for puppet," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 416–430. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908083

[7] J. Wettinger, U. Breitenbücher, and F. Leymann, "Compensation and convergence — comparing and combining deployment automation approaches," *International Journal of Cooperative Information Systems*, vol. 24, no. 03, p. 1541001, 2015. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0218843015410014

[8] 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.

[9] 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.

[10] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An empirical analysis of the docker container ecosystem on github," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 323–333. [Online]. Available: https://doi.org/10.1109/MSR.2017.67

[11] A. K. Jha, S. Lee, and W. J. Lee, "Developer mistakes in writing android manifests: An empirical study of configuration errors," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 25–36. [Online]. Available: https://doi.org/10.1109/MSR.2017.41

[12] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, Oct 2009, pp. 390–400.

[13] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka, "Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains," in *2013 IEEE International Conference on Software Maintenance*, Sept 2013, pp. 260–269.

[14] B. D. Bois, S. Demeyer, and J. Verelst, "Refactoring - improving coupling and cohesion of existing code," in *11th Working Conference on Reverse Engineering*, Nov 2004, pp. 144–151.

[15] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ser. ICSM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 350–359. [Online]. Available: http://dl.acm.org/citation.cfm?id=1018431.1021443

[16] [Online]. Available: https://docs.chef.io/ruby.html

[17] [Online]. Available: https://github.com/pulseenergy/chef-style-guide

[18] [Online]. Available: https://docs.chef.io/

[19] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 368–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850947.853341

[20] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[21] [Online]. Available: http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html

[22] [Online]. Available: https://github.com/swc-rwth/InfrastructureAsCodeSmells

[23] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1998.

[24] P. Jalote, *An Integrated Approach to Software Engineering*, 3rd ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.