Towards a Catalogue of Software Quality Metrics for Infrastructure Code

Stefano Dalla Palma[a], Dario Di Nucci[a], Fabio Palomba[b], Damian A. Tamburri[a]

[a] *Tilburg University - Jheronimus Academy of Data Science, 's-Hertogenbosch, The Netherlands*
[b] *University of Salerno, Fisciano, Italy*

Abstract

Infrastructure-as-code (IaC) is a practice to implement continuous deployment by allowing management and provisioning of infrastructure through the definition of machine-readable files and automation around them, rather than physical hardware configuration or interactive configuration tools. On the one hand, although IaC represents an ever-increasing widely adopted practice nowadays, still little is known concerning how to best maintain, speedily evolve, and continuously improve the code behind the IaC practice in a measurable fashion. On the other hand, source code measurements are often computed and analyzed to evaluate the different quality aspects of the software developed. However, unlike general-purpose programming languages (GPLs), IaC scripts use domain-specific languages, and metrics used for GPLs may not be applicable for IaC scripts. This article proposes a catalogue consisting of 46 metrics to identify IaC properties focusing on Ansible, one of the most popular IaC language to date, and shows how they can be used to analyze IaC scripts.

Keywords: Infrastructure as Code; Software Metrics; Software Quality.

1. Introduction

The information technology market is increasingly focused on "need for speed": speed in deployment, faster release-cycles, speed in recovery, and more. This need is reflected in DevOps, a family of techniques that shorten the software development cycle and intermix software development activities with IT operations [1, 2]. As part of the DevOps menu, Infrastructure-as-Code (IaC) [3] *"promotes managing the knowledge and experience inside reusable scripts of infrastructure code, instead of traditionally reserving it for the manual-intensive labor of system administrators which is typically slow, time-consuming, effort-prone, and often even error-prone"*.

While IaC represents an ever-increasing widely adopted practice [2–4], still little is known concerning how to maintain best, speedily evolve, and continuously improve the code behind the IaC practice. Nevertheless, it is picking up more and more traction in different domains [5–7].

A recent survey by Guerriero et al. [8], conducted with industrial practitioners and experts, raised concerns about code quality and explicitly mentioned the need for instruments to support them when developing infrastructure code. Among others, the authors reported insights regarding IaC-specific tools and languages that are common among practitioners. Currently, the IaC ecosystem is characterized by a plethora of different and often overlapping (in terms of their goals) tools and languages. Thus, it is crucial to study their adoption to identify the de-facto standard ways to write IaC. This analysis showed that no IaC tool is used by more than 60% of respondents, with Ansible being the second most used technology (with 52.2% of respondents using it, below the containerization technology Docker), confirming it as the de-facto standard configuration management technology.

In this context, we believe that the definition of source code metrics able to model the quality aspects of IaC could enable DevOps engineers to maintain and evolve them during Quality Assurance activities effectively.

In this article, we propose a new catalog composed of 46 measures that identify quality IaC code properties for Ansible[1], and showcase how they can be used to analyze infrastructure code. The advantages of a metrics-based quality management approach to infrastructure code are manifold, among others:

- The analysis of IaC properties can help developers understand and improve the quality of their infrastructure through incremental refactoring instead of the conventional trial-and-error approach;
- Source code properties can be used as early indicators of faulty infrastructure scripts potentially leading to expensive infrastructure failures[2];
- Specific metrics can be defined across IaC languages to understand the mutual and combined characteristics of Infrastructure-as-Code blends instead of focusing on a single vendor-locked IaC solution.

To the best of our knowledge, this work is the first to elicit a broad set of quality metrics that describe and

[1]`https://www.ansible.com/`
[2]`https://www.cloudcomputing-news.net/news/2017/oct/30/glitch-economy-counting-cost-software-failures/`

quantify the characteristics of infrastructure code to support DevOps engineers when maintaining and evolving it. However, it is worth noting that our catalog of metrics is broad in scope on purpose, to allow possible further studies on the relation between the proposed metrics and the quality of infrastructure code, posing the basis for a larger evaluation of infrastructure code quality.

The paper is structured as follows: in Section 2 we briefly describe the background information of this work. In Section 3 we describe the source code measures in detail. Finally, in Section 4, we conclude the article and present future research directions.

2. Background and Related Work

In this section, we briefly describe the background information to outline the context of this article and the previous works aimed at identifying source code properties that characterize IaC.

2.1. Ansible

The primary enabler for IaC has been the advent of cloud computing, which has made the programmatic provisioning, configuration, and management of computational resources more common. Subsequently, many different languages and corresponding platforms have been developed, each of which deals with a specific aspect of infrastructure management: virtual machines (e.g., Cloudify, Terraform), container technologies (e.g., Docker Swarm, Kubernetes), configuration management tools (e.g., Chef, Ansible, Puppet). Among them, Ansible is gaining traction in the last years as a simple and agent-less (i.e., no master node) alternative to other more complex IaC technologies such as Chef and Puppet[3].

Ansible is an automation engine based on the YAML language that automates cloud provisioning, configuration management, and application deployment. It works by connecting to nodes and pushing out scripts called *Ansible modules*. Most of which describe the state of the system. Then, Ansible executes and removes these modules when not needed.

However, while modules allow for the proper functioning of Ansible scripts, *playbooks* make possible the orchestration of multiple slices of the infrastructure topology, with exact control on the scalability of the architecture (e.g., how many machines to tackle at a time). Playbooks are essential for configuration management and multi-machine deployment in Ansible. They can declare configurations and orchestrate steps of any manual ordered process by launching tasks within one or more *plays*. The goal of a play is to map a group of hosts to some well-defined roles, represented by Ansible *tasks*, which in sum are calls to *Ansible modules*.

As an example, Figure 1 shows an Ansible code snippet representing a playbook that provisions and deploys a website[4]. Hence, it configures various aspects such as the ports to be opened on the host container, the name of the user account, and the desired database to be deployed. It first targets the web servers to ensure that Apache server is at the latest version. Then, it checks the database servers to ensure that postgreSQL is at the latest version, and that is started. It achieves this by mapping the hosts (lines 2 and 13) to their respective tasks (lines 8-11, 17-20, 22-25). There, `yum` and `service` are modules to manage packages with the yum package manager and to control services on remote hosts, respectively; `name` (i.e., the name of the package and the database) and `state` (i.e., whether present, absent, or otherwise) are parameters of these modules. In short, by composing a playbook of multiple plays, it is possible to orchestrate multi-machine deployments and run specific commands on the machines in the `webservers` group and in those in the `databases` group.

```yaml
 1  ---
 2  - hosts: webservers
 3    vars:
 4      http_port: 80
 5    remote_user: root
 6
 7    tasks:
 8    - name: ensure apache is at the latest version
 9      yum:
10        name: httpd
11        state: latest
12
13  - hosts: databases
14    remote_user: root
15
16    tasks:
17    - name: ensure postgresql is at the latest version
18      yum:
19        name: postgresql
20        state: latest
21
22    - name: ensure that postgresql is started
23      service:
24        name: postgresql
25        state: started
26
```

Figure 1: An example of Ansible code.

2.2. Related Work

Only a few research studies have been conducted on the current development practices of infrastructure code [8] or analyzed source code properties to evaluate the quality of Infrastructure as Code [9–11]. Most of the previous work

[3]Stemming from `https://github.com/search` using as search terms 'ansible', 'puppet' and 'chef'

[4]Adapted from Ansible documentation: `https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html` (Accessed April 2020)

describes code quality in terms of smelliness and defects-proneness of software components that are described via Chef and Puppet configuration management technologies.

Looking at code smells [12], Sharma et al. [9], Schwarz et al. [13], and Rahman et al. [14] applied the well-know concept to IaC. The properties identified by these studies can be grouped into: (i) *Implementation Configuration* such as complex expressions and deprecated statements; (ii) *Design Configuration* such as broken hierarchies and duplicate blocks [9]; (iii) *Security Smells* such as admin by default and hard-coded secrets [14]; (iv) *General Smells* such as long resources and too many attributes [13].

As for defect prediction, Rahman et al. [10] identified ten source code measures that significantly correlate with defective infrastructure as code scripts such as properties to execute bash and/or batch commands, to manage file permissions and SSH keys, to execute external scripts.

As for IaC quality in general, Bent et al. [11] explored the notion of code quality for Puppet code by performing a survey among Puppet developers and developed a measurement model for the maintainability aspect of Puppet code quality. They implemented the code quality model in a software analysis tool and validated their work by a structured interview with Puppet experts and by comparing the tool results with the quality judgments of those experts. They showed that the measurement model provides quality judgments of Puppet code that closely match the judgments of experts, which they deemed the model appropriate and usable in practice.

In this article, we build on this line of research to define source code measures able to further model quality aspects of IaC. We focus on Ansible, rather than Puppet and Chef, for a two-fold reason: (i) Ansible is the most popular IaC language on GitHub to date[3]; (ii) at the best of our knowledge, no attempts to analyze source code properties for Ansible has been previously made.

3. Measuring IaC Quality

Software measurement is a quantified attribute of a characteristic of a software product [15]. One method to perform software measurement is to use a set of metrics to analyze the code itself. Examples of metrics are the number of lines and functions in a single file and the number of files in an application.

3.1. Methodology

To extract the metrics, we applied the following methodology. Given the novelty of the topic, we first looked for traditional and language-agnostic source code metrics that are potentially applicable to IaC. We stemmed from a survey of almost 300 traditional and object-oriented source code metrics [16] such as executable, commented and blank lines of code, function count, class entropy complexity, and average method size). Needless to say that most of the object-oriented metrics do not apply to IaC: only 8 of them are present in our catalog. Then, some of the metrics applicable to IaC were introduced by Rahman et al. [10] for Puppet and ported to Ansible. Finally, we searched for metrics that are specifically inherent to IaC scripts written in Ansible, starting from the *atomic* structural characteristics described in the documentation[5] for which structural metrics were directly implementable. We moved towards the more complex ones that spread through multiple scripts and/or can be expressed as a combination of atomic measures. Those cover most of the constructs related to playbooks and tasks, such as plays, tasks, and modules, and include metrics dealing with error handling, bad and best practices (e.g., using deprecated statements and naming tasks uniquely, respectively), access of data from outside sources, and more.

The search process was performed by the first author of this paper, who scanned each resource to elicit an initial set of metrics that were possibly suitable for measuring IaC code quality. In this step, the author considered two selection criteria: (i) the metric must be elicitable from the source analyzed, e.g., when reading the Ansible documentation, the first author ensured that the description was accurate enough to enable the definition of a metric; (ii) the meaning of the metric must be clear to enable discussions on its potential impact on infrastructure code quality. Afterward, an open discussion and card-sorting exercise [17] with the remaining authors was enacted to refine the catalog and, whenever needed, assigning self-explainable names or scopes to the metrics.

More specifically, after the search, the authors opened a joint discussion on the resources retrieved. They went through the list and analyzed the metrics to establish a rationale and possible impact that each metric may have on the quality of infrastructure code. At this stage, they also assigned a name to the metrics: if a metric came from existing papers which already named it, they kept the same name; otherwise, they assigned a new, self-explainable name. Since this was an open discussion, cases of disagreement related to names or rationale of the metrics were immediately discussed and solved. In the second round, the authors proceeded by grouping metrics and discussing which of them may be considered as language-agnostic. Also in this case, the synchronous discussion allowed them to solve cases of disagreements immediately.

At the end of the process, it was possible to classify the initial set of metrics in three groups: (i) object-oriented metrics that can be ported to Ansible, and IaC in general; (ii) metrics that were investigated in previous works on IaC and that can be ported to Ansible, and therefore to similar languages; and (iii) metrics related to best and bad practices in Ansible (which are often reported in the documentation or external resources as books). The first two sets concern metrics that were investigated with respect to their value to code quality (even though some of them

[5]Ansible documentation [6]

Table 1: *Ansible metrics* – Summary of the identified and implemented metrics that identify source code properties

Measure	**Measurement techniques**	**Scope**
LinesBlank	Total empty lines	General
LinesComment	Count statements starting with `#`	General
LinesSourceCode	Total lines of executable code	General
NumCommands	Count of `command`, `expect`, `psexec`, `raw`, `script`, `shell`, and `telnet` syntax occurrences	General
NumConditions	Count of `is`, `in`, `==`, `!=`, `>`, `>=`, `<`, `<=` occurrences in `when`	General
NumDecisions	Count of `and`, `or`, `not` syntax occurrences in `when`	General
NumDeprecatedKeywords	Count the occurrences of deprecated keywords	General
NumEnsure	Count of ”`\w+.stat.\w+ is defined`” regex matches in `when`	General
NumFile	Count of `file` syntax occurrences	General
NumFileMode	Count of `mode` syntax occurrences	General
NumLoops	Count of `loop` and `with_*` syntax occurrences	General
NumMathOperations	Count of `+`, `-`, `/`, `//`, `%`, `*`, `**` syntax occurrences	General
NumParameters	Count the keys of the dictionary representing a module	General
NumPaths	Count of `paths`, `src` and `dest` syntax occurrences	General
NumRegex	Count of `regexp` syntax occurrences	General
NumSSH	Count of `ssh_authorized_key` syntax occurrences	General
NumSuspiciousComments	Count comments with `TODO`, `FIXME`, `HACK`, `XXX`, `CHECKME`, `DOCME`, `TESTME`, or `PENDING`	General
NumURLs	Count of `url` syntax occurrences	General
NumTokens	Count the words separated by a blank space	General
NumUserInteractions	Count of `prompt` syntax occurrences	General
NumVariables	Sum len(`vars`) in plays	General
TextEntropy	$-\sum_{t \in tokens(script)} p(t) log_2(p(t))$	General
NumPlays	Count of `hosts` syntax occurrences	Playbook
NumRoles	Length of the `roles` section in a playbook	Playbook
AvgPlaySize	LinesSourceCode(playbook)/NumPlays	Playbook | Tasks list
AvgTaskSize	LinesSourceCode(tasks)/NumTasks	Playbook | Tasks list
NumBlocks	Count of `block` syntax occurrences	Playbook | Tasks list
NumBlocksErrorHandling	Count of `block-rescue-always` section occurrences	Playbook | Tasks list
NumDeprecatedModules	Count the occurrences of deprecated modules	Playbook | Tasks list
NumDistinctModules	Count of *distinct* modules maintained by the community	Playbook | Tasks list
NumExternalModules	Count occurrences of modules not maintained by the community	Playbook | Tasks list
NumFactModules	Count occurrences of fact modules (listed in the doc)	Playbook | Tasks list
NumFilters	Count of `|` syntax occurrences inside {{*}} expressions	Playbook | Tasks list
NumIgnoreErrors	Count of `ignore_errors` syntax occurrences	Playbook | Tasks list
NumImportPlaybook	Count of `import_playbook` syntax occurrences	Playbook | Tasks list
NumImportRole	Count of `import_role` syntax occurrences	Playbook | Tasks list
NumImportTasks	Count of `import_tasks` syntax occurrences	Playbook | Tasks list
NumInclude	Count of `include` syntax occurrences	Playbook | Tasks list
NumIncludeRole	Count of `include_role` syntax occurrences	Playbook | Tasks list
NumIncludeTasks	Count of `include_tasks` syntax occurrences	Playbook | Tasks list
NumIncludeVars	Count of `include_vars` syntax occurrences	Playbook | Tasks list
NumKeys	Count of *keys* in the dictionary representing a playbook or tasks	Playbook | Tasks list
NumLookups	Count of **lookup(*)** occurrences	Playbook | Tasks list
NumNameWithVariables	Count of `name` occurrences matching the `".*{{\w+}}.*"` regex	Playbook | Tasks list
NumTasks	len(`tasks`) in playbook or tasks file	Playbook | Tasks list
NumUniqueNames	Count of `name` syntax occurrences with unique values	Playbook | Tasks list

not yet studied in the context of infrastructure code); the latter set emerged when analyzing the recommendations to design quality infrastructure code.

3.2. A Catalogue of Metrics for IaC

Our catalog is composed of 46 code metrics. In particular, (i) 8 metrics are related to language-agnostic code characteristics, (ii) 14 metrics have been adapted by those previously developed by Rahman et al. [10] for Puppet, (iii) 24 metrics concerns some inherent characteristics of Ansible that are observable in other orchestration configuration languages as well. The metrics related to the first group are discussed in the following. They all concern the characterization of long/complex infrastructure code. As widely reported in the literature on source code quality [18, 19], those metrics could potentially make the code more prone to be defective. While no empirical evaluation of the impact of these metrics is still available in the context of IaC, we hypothesize that similar conclusions could be reached.

- LinesSourceCode, LinesComment, and LinesBlank to count the total number of *executable lines of code*, *lines of comments*, and *blank lines*, respectively. The example in Figure 1 has 20 lines of code, no comments, and four blank lines. *Interpretation:* the more the lines of code, the more complex and the more challenging to maintain the blueprint.

- NumConditions and NumDecisions where a *condition* is a Boolean expressions containing no Boolean operators (e.g., `and` and `or`) and a *decision* a Boolean expression composed of conditions and one or more Boolean operators. The example in Figure 1 has neither conditions nor decisions. In Ansible, those are mostly specified through the `when` statement that is not present in this case. *Interpretation:* the more the conditions and decisions, the more complex and the more challenging to maintain the blueprint.

- TextEntropy to measure the complexity of a script based on its information content, analogous to the *class entropy complexity. Interpretation:* the higher the entropy, the more challenging to maintain the blueprint.

- NumTasks to measure the number of functions in a script, analogous to the traditional Number of Methods Call [16]. We consider an Ansible *task* equivalent to a method, as its goal is to execute a module with very specific arguments. The example in Figure 1 has three tasks, so NumTasks=3. *Interpretation:* the higher the number of tasks, the more complex and the more challenging to maintain the blueprint.

- AvgTaskSize analogous to the Average Method Complexity [16], to measure the average size of program modules. The example in Figure 1 has three tasks. Therefore AvgTaskSize=4 (rounded to the nearest unit). *Interpretation:* the higher the more complex and the more challenging to maintain the blueprint.

Follows the second group of metrics that we generalized from the previous work conducted by Rahman et al. [10], where the authors observed a significant correlation with defective infrastructure as code scripts. Specifically, they conducted a qualitative analysis with practitioners and empirically validated such metrics in the scope of defect prediction of Puppet code, and a common interpretation of the following metrics is that the higher their value, the more prone to defects the blueprint:

- NumCommands – Puppet allows developers to execute external commands via the resource type `exec`. For the same functionality, Ansible provides several modules: `command`, `expect`, `psexec`, `raw`, `script`, `shell`, and `telnet`. *Interpretation:* the number of external commands makes the blueprint more complex and more challenging to maintain.

- NumEnsure – `ensure` is a Puppet source code property used to check the existence of a file, directory or symbolic links. In Ansible, the existence of those entities can be checked through the module `stat`. *Interpretation:* blueprints with many file existence checks are less prone to misbehaviour but more challenging to test.

- NumFile – `file` is a source code property used to manage files, directories, and symbolic links. It exists either in Puppet (as a resource type) and Ansible (as a module). *Interpretation:* the higher this metric, the more challenging to maintain and test the blueprint.

- NumFileMode – `mode` is a source code property used to set permissions of files. It exists either in Puppet (as an attribute of the `file` resource type) and Ansible (as a parameter of the `file` module). *Interpretation:* higher levels of this metric make the blueprint less prone to misbehaviour but more challenging to test.

- NumInclude – In Puppet, other scripts can be executed with the `include` function. This functionality in Ansible is provided by several `include` and `import` modules that allow users to break up large playbooks into smaller files, which can be used across multiple playbooks or even multiple times within the same playbook. `Import` statements are pre-processed at compilation-time:

  – NumImportPlaybook – `import_playbook` is used to include a file with a list of plays to be executed in the current playbook.
  – NumImportRole – `import_role` is used to load a *role* when the playbook is parsed.
  – NumImportTasks – `import_tasks` is used to import a list of tasks to be added to the current playbook for subsequent execution.

  `Include` statements are processed at execution-time[7]:

  – NumInclude –`include` is used to include a file with a list of plays or tasks to be executed in the current playbook.
  – NumIncludeRole – `include_role` is used to dynamically load and execute a specified role as a task.
  – NumIncludeTasks – `include_task` is used to include a file with a list of tasks to be executed in the current playbook.
  – NumIncludeVars – `include_vars` is used to load YAML or JSON variables from a file or directory, recursively, during task run-time.

  Interpretation: on the one hand, the more the includes and imports, the higher the reusability. On the other hand, the more the includes and imports, the more difficult to maintain the blueprint.

[7]https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_includes.html

- NumParameters – In Puppet, the state of a resource is described with an attribute. Similarly, in Ansible *parameters* (or arguments), describe the desired state of the system. *Interpretation:* the more the parameters, the more challenging to debug and test the blueprint.

- NumSSH – `ssh_authorized_key` is a Puppet source code property used to manage SSH authorized keys. The analog in Ansible is the module `authorized_key`, used to add or remove SSH authorized keys for particular user accounts. *Interpretation:* the more this property, the more challenging to maintain and test the blueprint.

- NumURLs – URL refers to URLs used to specify a configuration. Ansible defines a module called `uri` to interact with *http* and *https* web services, and requires to set a parameter `url`. *Interpretation:* the more the URLs, the more challenging to maintain and test the blueprint.

Figure 2 shows a code snippet on which we computed such metrics. In this example, NumInclude = 1 (line 2), NumParameters = 3 (lines 6, 11, 12), NumFile = 1 (line 10), NumFileMode = 1 (line 12), and NumEnsure = 1 (line 13); NumSSH = 0; NumURLs = 0. It also has two conditions bound by a logic `and` at line 11, consequently NumConditions = 2 and NumDecisons = 1.

```
 1   - name: Perform some preliminar tasks
 2     import_tasks: stuff.yaml
 3
 4   - name: Retrieving file status
 5     stat:
 6       path: /etc/foo.conf
 7     register: conf
 8
 9   - name: Change file permission if exists
10     file:
11       path: /etc/foo.conf
12       mode: '0644'
13     when: conf.stat.exists is defined and conf.stat.exists
```

Figure 2: Some of the source code properties designed by Rahman et al. [10] for Puppet and ported to Ansible

The third group of metrics was derived from the Ansible documentation and are mainly related to best and bad practices and (external) data management. These following metrics could affect the quality of infrastructure code in terms of comprehensibility and maintainability:

- DeprecatedKeywords and DeprecatedModules – Deprecated modules and keywords usage is discouraged as they are kept for backward compatibility only. *Interpretation:* the more the deprecated keywords and modules, the more difficult it is to maintain and evolve the code. In addition, the higher the likelihood to crash if the current system does not support retro-compatibility.

- NumBlocks and NumBlocksErrorHandling – A block logically groups tasks within a section, but also allows for exception handling by appending a `rescue` or an `always` to the block. The tasks in the block are typically executed. A `rescue` section is executed only when an error is raised, while an `always` section is executed in any case (i.e., included in case of errors). *Interpretation:* the more the blocks, the easier code maintenance. However, many blocks without rescue/always sections increase the system's chance to crash when incurring in wrong behaviours.

- NumDistinctModules – Modules are reusable and standalone scripts called by tasks. They allow to change or get information about the state of the system and can be interpreted as a degree of responsibility of the blueprint. Therefore, we hypothesize that a blueprint consisting of many distinct modules is less self-contained and potentially affect the complexity and maintainability of the system, as it is responsible for executing many different tasks rather than a task several times with different options, for example, to ensure the presence of dependencies in the system (as the `yum` module in Figure 1).
Interpretation: the more the distinct modules, the more challenging to maintain the blueprint.

- NumExternalModules – Many modules are maintained by the community. However, users can create and maintain their modules, called *external modules*. While modules maintained by the community require to be fully documented and tested, this is not needed for external modules. Therefore, we conjecture that a blueprint with a high number of external modules is more challenging to maintain than a blueprint containing only modules maintained by the community. *Interpretation:* The more the external modules, the more challenging to maintain the blueprint and the higher the chance of system's misbehaviour.

- NumFactModules – *Fact modules* are modules that do not change the state of the system but only return data. We hypothesize that blueprints with many fact modules are less prone to unexpected behaviours and easier to test, as they do not alter the system's state. *Interpretation*: the more the `fact` modules the easier to test and maintain the blueprint.

- NumFilters – Filters transform the data of a template expression, for example, for formatting or rendering them. Although they allow for transforming data in a very compact way, filters can be concatenated to perform a sequence of different actions, as shown in Figure 3. We believe that this aspect may potentially affect the readability and maintainability of the code. *Interpretation:* the more the filters, the lower the readability and the more challenging to maintain the blueprint.

- NumIgnoreErrors – Ansible provides different ways to handle errors. Among others, it is possible to prevent a playbook from stopping when a task fails by setting `ignore_errors:True`. However, ignoring errors is considered as a bad practice, since it hides error handling. *Interpretation:* the more the `ignore_errors:True` statements, the more the system is hard to debug and prone to misbehaviour.

- NumLookups – Lookups are an advanced feature that allows access to outside data sources and requires an advanced working knowledge of Ansible plays before incorporating them. Some lookups pass arguments to a shell, and one should use them carefully to ensure safe usage. *Interpretation:* the more the lookups, the higher the chance of system's misbehaviour and the more challenging to maintain.

- NumSuspiciousComments – Suspicious comments warn the presence of defects, missing functionality, or the system's weakness. *Interpretation:* the more the suspicious comments, the higher the chance of system's misbehaviour because of missing or incomplete functionalities.

- NumUniqueName – Uniquely naming plays and tasks is a best practice to locate problematic tasks quickly. Duplicate names may lead to not deterministic or at least not obvious behaviours [20]. *Interpretation:* the more the entities with unique names the higher the maintainability and readability of the blueprint.

- NumNamesWithVariables – Having uniqueness as a goal, many playbook developers prefer to use variables instead of hard-coding names. This strategy may work well, but authors need to take care of the source of the variable data they are referencing. Indeed, variable data can come from various locations, and the values assigned to variables can be defined many times. For the sake of play and task names, only variables for which the values can be determined at playbook parse time will parse and render correctly. If the data of a referenced variable is discovered via a task, the variable string will be displayed unparsed in the output [20], potentially affecting debugging and software auditing. *Interpretation:* although names with variable could make more succinct code, they could hinder code debugging and potentially lead to system's misbehaviour.

- NumUserInteractions – In some cases, an Ansible script requires the user input (e.g., username and password to access a service). Asking for external input may potentially affect the correctness of the program at run-time. User interactions have to be handled by the program with several conditions. We conjecture that, if not handled properly, a given input may lead the system to crash at run-time. *Interpretation:* the more user interactions, the higher the chance of system's misbehaviour.

```yaml
- name: generate multiple hostnames
  debug:
    msg: "{{['foo','bar']|product(['com'])|map('join', '.')|join(',')}}"

# This would result in:
# { "msg": "foo.com,bar.com" }

```

Figure 3: The product filter returns the cartesian product of the input iterables, that is roughly equivalent to nested for-loops.

The remaining metrics are self-describing and measure different aspects of the size of a blueprint which may affect its quality in terms of complexity and readability: NumPlays, AvgPlaySize, NumRoles, NumVariables, NumLoops, NumMathOperations, NumPaths, NumRegex (i.e., regular expressions), NumTokens (i.e., words separated by blank spaces), and NumKeys (i.e., keys of the dictionary representing a playbook or a list of tasks). In particular, paths and regular expressions are often subject to typos, which might lead to run-time errors if they are not correctly handled. We conjecture that the more they are, the higher the chance the system will run into unexpected behaviour. In general, *we hypothesize that the higher the number of the aforementioned source code properties, the more complex the blueprint.*

The complete list of the metrics we designed is shown in Table 1. The code metrics can be categorized in terms of their scope (i.e., *general*, *playbook*, and *tasks list*) depending on the particular construct and/or artifact they target. *General* metrics can apply to either playbooks and task files, i.e., files that contain a flat list of tasks, but can also generalize to other languages. *Playbook* metrics operate within a single playbook; while *tasks list* metrics can operate either within a playbook (when the construct *tasks* is present) and within tasks files.

Figure 4 shows an Ansible code snippet that consists of a list of tasks adapted from the Ansible documentation[8]. The tasks are grouped in two blocks: the first block (lines 3-8) is used to handle errors along with the `rescue` and `always` sections (lines 8 and 11); the second block (lines 16-27) is used to allow the tasks' execution only if the current system distribution is 'CentosOS' (line 27). Therefore, NumBlocks = 2 and NumBlocksErrorHandling = 1. The `ignore_errors:True` at line 21 prevents a playbook from stopping if something goes wrong with the Apache server's installation. The error should not be ignored but caught and handled in a `rescue` section. The word `TODO` in the comment is suspicious and

[8]`https://docs.ansible.com/ansible/latest/user_guide/playbooks_blocks.html(AccessedonApril2020)`

```
1   - name: Attempt and graceful roll back demo
2     block:
3       - debug:
4           msg: 'I execute normally'
5       - name: I force a failure
6         command: /bin/false
7     rescue:
8       - debug:
9           msg: 'I caught the failure'
10    always:
11      - debug:
12          msg: 'I always execute :-)'
13
14  - name: Install and start Apache
15    block:
16      - name: install httpd
17        yum:
18          name: httpd
19          state: latest
20        ignore_errors: True # TODO: handle error in rescue
21
22      - name: start service bar and enable it
23        service:
24          name: bar
25          state: started
26    when: ansible_facts['distribution'] == 'CentOS'
```

Figure 4: An example of properties inherent to Ansible.

suggests a possible issue. Here NumIgnoreErrors = 1 and NumSuspiciousComments = 1. Other information that can be extracted from the snippet relate to the number of distinct modules, namely `debug`, `command`, `yum` and `service`, and the the number of unique names (lines 2, 6, 15, 17, 23). Note that the `name` at lines 19 and 25 are not tasks' names, but module parameters. The metrics calculated here are NumDistinctModules = 4 and NumUniqueNames = 5.

4. Conclusion and Research Roadmap

Infrastructure code is emerging as the de-facto challenge for software maintenance and evolution of the coming years, especially concerning complex systems blending microservices with serverless components. Indeed, varied and elaborate infrastructure designs may well change dynamically during operations. On the one hand, infrastructure source code properties may be used and combined as surrogate metrics for defect-proneness of infrastructure components and to identify smells and refactoring operations during Quality Assurance activities connected to infrastructure code. On the other hand, the existing code measures cannot currently and directly model the aforementioned aspects of IaC.

We propose a broad catalogue of 46 structural-based code measures to evaluate the different aspects of IaC, the most comprehensive measures set for IaC to the best of our knowledge. Although our implementation targets particularly Ansible, the aforementioned measures are equivalent (and portable) to other languages (e.g., Chef, Puppet) and offer a general-purpose metrics-based approach for IaC quality evaluation. Given the early-stage research on IaC analytics, there are several avenues for future work.

Empirically Investigating the Relationship between IaC Metrics and Code Quality. We are aware that some of the proposed metrics may have little effect on code quality. To accurately identify and validate what metrics significantly affect a given quality aspect of a configuration management system, future empirical studies are required. First, more research is needed to understand the relation between the presented metrics and the quality of IaC blueprints. Second, there is still a lack of empirical evidence regarding the scope and usability of such measures, e.g., whether they can be used and/or combined to detect code smells and bugs. A step in this direction is the evaluation of the proposed metrics as a proxy of a defect prediction model for IaC scripts. A further step would be the mapping of the proposed metrics to the software quality attributes they actually model (e.g., maintainability, complexity, reusability).

IaC Metrics in Software Defect Prediction. Software Defect Prediction [21] helps to identify the parts of the systems that may require more testing because prone to defects. The definition of effective prediction of defect-prone blueprints in the Infrastructure-as-Code scope may help DevOps engineers focus on such demanding blueprints during testing and maintenance activities, thus allocating effort and resources more efficiently. Product and process metrics (i.e., metrics that capture aspects concerning the development process rather than the code itself, for example the number of modification to a file) have been widely adopted in defect prediction of systems written in General Purpose Languages, and in many cases process metrics over-performed product metrics in terms of classification performance [22, 23]. Nevertheless, little is known about the prediction power of both sets of metrics across domain-specific languages such as IaC. As opposed to product metrics, process metrics are language-agnostic and do not need to be ported to IaC (i.e., the original ones can be used). Therefore, the definition of the proposed catalog would allow for a fair comparison between traditional process metrics and IaC-oriented product metrics in the scope of defect prediction of infrastructure code.

IaC Metrics Generalization. Afterwards, we plan to generalise our catalogue to support other configuration orchestration languages. To this aim, we will map the Ansible-specific characteristics to other languages. Among the others, we plan to support the de-iure industry standard for infrastructure code, namely, the ”Topology and Orchestration Specification for Cloud Applications” TOSCA [24] language, a YAML-based OASIS standard for defining infrastructure topologies. This specification was initially designed as an open standard for formatting templates. So, tasks (e.g., cloud resource deployment and orchestration) could be translated into a generally readable form and become more portable across platforms. Overall, the standard aims to make it easier to update, extend, or move cloud-based resources, thus opening up opportunities for building a universal software-quality model for configuration orchestration languages.

Acknowledgements

The authors would like to thank the colleagues and anonymous reviewers for their invaluable feedback. Stefano, Dario, and Damian are supported by the European Commission grant no. 825040 (H2020 RADON). Fabio acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PZ00P2_186090 (TED).

References

[1] L. Bass, I. Weber, L. Zhu, DevOps: A software architect's perspective, Addison-Wesley Professional, 2015.

[2] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, D. A. Tamburri, Devops: introducing infrastructure-as-code, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, 2017, pp. 497–498.

[3] K. Morris, Infrastructure as code: managing servers in the cloud, " O'Reilly Media, Inc.", 2016.

[4] M. Hüttermann, Infrastructure as code, in: DevOps for Developers, Springer, 2012, pp. 135–156.

[5] M. Jarschel, Network function virtualization: Towards the commoditization of middle boxes (11 2013).

[6] D. Soldani, B. Barani, R. Tafazolli, A. Manzalini, I. Chih-Lin, Software defined 5g networks for anything as a service [guest editorial], IEEE Communications Magazine 53 (9) (2015) 72–73.

[7] P. Lipton, D. Palma, M. Rutkowski, D. A. Tamburri, Tosca solves big problems in the cloud and beyond!, IEEE cloud computing (2018).

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

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

[10] A. Rahman, L. Williams, Source code properties of defective infrastructure as code scripts, Information and Software Technology 112 (2019) 148–163.

[11] E. Van der Bent, J. Hage, J. Visser, 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), IEEE, 2018, pp. 164–174.

[12] M. Folwer, Refactoring: Improving the design of existing programs (1999).

[13] J. Schwarz, A. Steffens, H. Lichter, Code smells in infrastructure as code, in: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), IEEE, 2018, pp. 220–228.

[14] A. Rahman, C. Parnin, L. Williams, The seven sins: Security smells in infrastructure as code scripts, in: Proceedings of the 41st International Conference on Software Engineering, 2019, pp. 164–175.

[15] N. Fenton, J. Bieman, Software metrics: a rigorous and practical approach, CRC press, 2014.

[16] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, C. Soubervielle-Montalvo, Source code metrics: A systematic mapping study, Journal of Systems and Software 128 (2017) 164–197.

[17] K. M. Lewis, P. Hepburn, Open card sorting and factor analysis: a usability case study., The Electronic Library 28 (3) (2010) 401–416.
URL `http://dblp.uni-trier.de/db/journals/el/el28.html#LewisH10`

[18] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: a benchmark and an extensive comparison, Empirical Software Engineering 17 (4-5) (2012) 531–577.

[19] H. Zhang, An investigation of the relationships between lines of code and defects, in: 2009 IEEE International Conference on Software Maintenance, IEEE, 2009, pp. 274–283.

[20] J. Keating, Mastering Ansible, Packt Publishing Ltd, 2015.

[21] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Transactions on Software Engineering 38 (6) (2011) 1276–1304.

[22] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: Proceedings of the 30th international conference on Software engineering, 2008, pp. 181–190.

[23] F. Rahman, P. Devanbu, How, and why, process metrics are better, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 432–441.

[24] C. N. Matt Rutkowski, Chris Lauwers, C. Curescu, Tosca simple profile in yaml version 1.3 (2019).
URL `http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html`