The Do's and Don'ts of Infrastructure Code: a Systematic Grey Literature Review

Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Damian Andrew Tamburri, Willem-Jan van den Heuvel

Jheronimus Academy of Data Science, The Netherlands

Fabio Palomba

University of Salerno, Italy

Abstract

Context. Infrastructure-as-code (IaC) is the DevOps tactic of managing and provisioning software infrastructures through machine-readable definition files, rather than manual hardware configuration or interactive configuration tools.
Objective. From a maintenance and evolution perspective, the topic has picked the interest of practitioners and academics alike, given the relative scarcity of supporting patterns and practices in the academic literature. At the same time, a considerable amount of grey literature exists on IaC. Thus we aim to characterize IaC and compile a catalog of best and bad practices for widely used IaC languages, all using grey literature materials.
Method. In this paper, we systematically analyze the industrial grey literature on IaC, such as blog posts, tutorials, white papers using qualitative analysis techniques.
Results. We proposed a definition for IaC and distilled a broad catalog summarized in a taxonomy consisting of 10 and 4 primary categories for best practices and bad practices, respectively, both language-agnostic and language-specific

Email addresses: i.p.k.weerasingha.dewage@tue.nl, m.garriga@uvt.nl, angeluromeu88@gmail.com, d.dinucci@uvt.nl, d.a.tamburri@tue.nl, w.j.a.m.vdnheuvel@uvt.nl (Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Damian Andrew Tamburri, Willem-Jan van den Heuvel), fpalomba@unisa.it (Fabio Palomba)

ones, for three IaC languages, namely Ansible, Puppet, and Chef. The practices reflect implementation issues, design issues, and the violation of/adherence to the essential principles of IaC.

Conclusion. Our findings reveal critical insights concerning the top languages as well as the best practices adopted by practitioners to address (some of) those challenges. We evidence that the field of development and maintenance IaC is in its infancy and deserves further attention.

Keywords: Infrastructure-as-code, DevOps, Grey Literature Review.

1. Introduction

The current information technology (IT) 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, 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.

IaC represents a widely adopted practice [3, 4, 5]. However, little is known concerning its code maintenance, evolution, and continuous improvement in academic literature, despite increasing traction in most if not all domains of society and industry: from Network-Function Virtualisation (NFV) [6] to Software-Defined Everything [7] and more [8].

However, little academic literature exists on infrastructure code since research on IaC is still in its early phases. At the same time, companies are working day-by-day on their infrastructure automation, as also witnessed by the considerable amount of grey literature on the topic. Hence, we can observe a gap between academic research and industry practices, mainly to figure out the technical, operational, and theoretical underpins and the best and bad practices when

developing IaC in the most popular languages.

This paper aims at addressing this gap with a systematic grey literature review. We shed light on the state of the practice in the adoption of IaC by analyzing 67 high-quality sources and the fundamental software engineering challenges in the field. In particular, we investigate: (1) how the industrial researchers and practitioners characterize IaC, and (2) the best/bad practices during general (i.e., language-agnostic) and language-specific (e.g., Puppet, Chef, Ansible) IaC development.

Specifically, we derived a more rigorous definition for infrastructure code. We identified a taxonomy consisting of 10 primary categories for best practices and 4 for bad practices, with all practices reflecting key improvements for DevOps processes around IaC. The practices reflect a range of scenarios: (a) implementation issues (e.g., naming convention, style, formatting, and indentation), (b) design issues (e.g., design modularity, reusability, and customizability of the code units of the different languages); (c) violation of/adherence to the essential principles of IaC (idempotence of configuration code, separation of configuration code from configuration data, and infrastructure/configuration management as software development). Overall, these issues highlight that the field of development and maintenance of IaC deserves further attention and further experimentally-proven tooling.

The rest of this paper is organized as follows. Section 2 provides a background on IaC. Section 3 poses our motivation and problem statement based on related work in the field. The research design and the research questions are presented in Section 4, while the results are provided in Section 5. In particular, Section 5.1 summarizes the selected sources; Section 5.2 focuses on IaC definition, classification, and features; Section 5.3 and Section 5.4 present best and bad practices of IaC development. We discuss the implications of our findings in Section 6 and the threats to validity in Section 7. Finally, Section 8 concludes the paper.

2. Background

```
- name: Configure webservers
  hosts: webservers
  roles:
    - web
- name: Configure databases
  hosts: dbservers
  roles:
    - db                playbook
```

```
[webservers]
  webs1
[dbservers]
  dbs1
inventory
```

```
- name: Install httpd                  role "web"
  yum:  name=httpd  state=latest
  when: ansible_os_family == 'RedHat'

- name: Install apache2
  apt: name=apache2 state=latest
  when: ansible_os_family == 'Debian'
```

```
- name: Create database user           role "db"
  mysql_user: user=myuser password=mypass

- name: Create database
  mysql_db: db=mydb state=present
```

(a)

```
class uniapache {
  if $::osfamily == 'RedHat' {
    $apachename   = 'httpd'
  }
  elsif $::osfamily == 'Debian' {
    $apachename   = 'apache2'
  }
  package { 'apache':
    name => $apachename,
    ensure => 'present',
  }
}
node 'webs1' {
  class { 'uniapache': }
}
node 'dbs1' {
  mysql::db { 'mydb':
    user     => 'myuser',
    password => 'mypass',
    host     => '127.0.0.1'
  }
}
```

(b)

```
node["lamp_stack"]["webs1"].each do
  case node[:platform]
  when 'RedHat'
    apachename = 'httpd'
  when 'Debian'
    apachename = 'apache2'
  end
  package apachename do
    action :install
  end
end

node["lamp_stack"]["dbs1"].each do
  mysql_database 'mydb' do
    connection(
        :host     => '127.0.0.1',
        :username => 'myuser',
        :password => 'mypass'
    )
    action :create
  end
end
```

(c)

Figure 1: Snippets of IaC scripts for installing Apache and MySQL in (a) Ansible, (b) Puppet, and (c) Chef

Infrastructure-as-code (IaC) is a process of managing and provisioning computing environments in which software systems will be deployed and managed, through the reusable scripts of infrastructure code [3]. In this section, we briefly introduce the three IaC technologies considered in this paper: Ansible [9], Puppet [10], and Chef [11]. We consider these IaC technologies because they are the most popular languages amongst practitioners according to our previous survey [12]. Below we present only a subset of the constructs of each IaC language, and will introduce the other constructs when we discuss about the best and bad practices of each language.

2.1. Ansible

In Ansible, a *playbook* defines an IT infrastructure automation workflow as a set of ordered *tasks* over one or more *inventories* consisting of managed infrastructure nodes. A *module* represents a unit of code that a task invokes. A module serves a specific purpose, for example, creating a MySQL database and installing an Apache webserver. A *role* can be used to group a cohesive set of tasks and resources that together accomplish a specific goal, for example, installing and configuring MySQL.

Figure 1(a) shows an Ansible snippet for setting up a MySQL database and an Apache webserver. The two roles *web* and *db* define the required tasks such as *Install httpd* and *Create database*. Each task uses a module to achieve its objective, for example, the task *Install httpd* uses the module *yum*. The inventory file defines the webserver node *webs1* and the database server node *dbs1*. The playbook applies the two roles to the two nodes.

2.2. Puppet

In Puppet, there are four key types of building blocks: (i) *Resources*, (ii) *Classes*, (iii) *Manifests*, and (iv) *Modules*. *Resources* define the properties of the system components (e.g., files, users, and groups) managed in a node, while *classes* are collection of *resources*. Puppet files containing definitions or declarations of Puppet *classes* are called *manifests*. Finally, *modules* are reusable and shareable units of Puppet codes performing a specific infrastructure automation task. A module can include artefacts such as manifests and other configuration files.

Figure 1(b) shows a Puppet snippet. The class *uniapache* uses the resource *package* to install the Apache web server. The node *webs1* declares this class to add its resources to the node. while the *dbs1* node creates a MySQL database employing the resource *mysql::db*.

2.3. Chef

In Chef, a *cookbook* represents an IT automation workflow. It consists of a set of *recipes*, which are a collection of *resources* to be created and managed on a node. A *resource* declares a system component and the actions to create and manage the component, for example, installing the package Apache. Chef recipes are written using Ruby.

Figure 1(c) shows a Chef recipe with two resources. The resource *package* is used to install Apache web server on the node *webs1*, and the resource *mysql_ database* is applied to create a MySQL database on the node *dbs1*.

3. Related Work and Research Goals

In this section, we discuss the related work and set forth our research goals.

3.1. Prior Research on IaC

Rahman et al. [13] recently performed a mapping study on IaC research and classified the existing work into several categories, which we summarize by providing an overview on framework and tools, empirical studies, and antipattern catalogs for IaC. Finally, we provide a summary of previous literature reviews on IaC-close topics.

3.1.1. Framework and Tool for IaC

The mapping study reported several tools/frameworks that extend the functionality of IaC for assessing and improving the quality of IaC scripts. Recently, Wurster et al. [14] developed TOSCA Lightning. This integrated toolchain enables modeling the deployment topology of an application using a subset of the TOSCA (Topology and Orchestration Specification for Cloud Applications) standard and converting such models into the artifacts used by production-ready deployment technologies such as Kubernetes and Ansible. Similarly, to support the quality assurance of IaC artifacts, several tools have been recently proposed. Dalla Palma et al. [15, 16, 17] proposed a set of tools to calculate quality metrics for Ansible scripts and projects and use them for predicting defective scripts. Kumara et al. [18] proposed a tool to detect smells in TOSCA scripts using an ontology-based approach. Borovits et al. [19] developed a tool to predict linguistic antipatterns in IaC using deep learning. Cito et al. [20] detected violations of Docker best practices, while Dai et al. [21] leveraged static code analysis and rule-based reasoning to detect risky IaC artifacts. Finally, Sotiropoulos et al. [22] crafted a tool to identify missing dependencies and notifiers in Puppet manifests by analyzing system call traces.

3.1.2. Empirical Studies related to IaC

According to the mapping study, IaC has been used to support the automated provisioning and deployment of applications on different infrastructures and

implement DevOps and continuous deployment. Several empirical studies focus on testing and quality assurance and the evolution of IaC artifacts to analyze how practitioners adopt this technology. IaC has been used to support the automated provisioning and deployment of applications on different infrastructures and implement DevOps and continuous deployment. Guerriero et al. [12] identified further insights on the challenges related to the IaC development and testing in industrial contexts by surveying 44 practitioners. Sandobalín et al. [23] focused on the effectiveness of IaC tools, while Rahman et al. [24, 25] on testing and security practices mined from grey literature. With similar goals, the latter analyzed the development practices that contribute to defective IaC scripts [26] and replicated previous studies [27]. Finally, Opdebeeck et al. [28] analyzed the adoption of semantic versioning in Ansible roles, while Kokuryo et al. [29] examined the usage of imperative modules in the same language.

3.1.3. Antipattern and Practices Catalogs for IaC

No previous systematic literature analyzed good and bad practices that developers adopt when implementing IaC. Although no studies defined and characterize the concept of IaC using gray literature, some previous work [30, 31] leverage grey literature to compile code smells catalogs for different languages. In particular, they relied on language style guides (e.g., Puppet style guide) and smell detection rules implemented in linters (e.g., Puppet-Lint) Sharma et al. [30] developed a catalog of design and implementation smells for Puppet. These are well-known violations of best practices for configuration code identified by analyzing the official documentation, the validation rules of Puppet-Lint, a few blog entries, and some video tutorials. Similarly, Schwarz et al. [31] compiled a catalog of smells for Chef, containing violations against the best practices for Chef, extracted from the official documentation. Kumara et al. [18] presented ten security and implementation smells for deployment models codified in TOSCA (Topology and Orchestration Specification for Cloud Applications). Using Docker Linter, Schermann et al. [20] detected violations of Docker best practices in open-source projects. Van der Bent et al. [32] defined a measurement model to

assess Puppet code quality. These metrics reflect the best practices and their violations in Puppet. Rahman et al. [33] derived a catalog of seven security smells in Puppet by analyzing such scripts in open-source repositories. Afterward, they developed a novel defect taxonomy [34] by analyzing defect-related commits, which includes eight defect categories, covering IaC source code, documentation, and IT infrastructure. Furthermore, they qualitatively analyzed Puppet-related questions that developers ask on StackOverflow [35]. The results of this analysis allowed them to identify 16 categories of questions that reflect key challenges and cover different aspects such as code syntax, testing, troubleshooting, security, resource provisioning, and software installation. In two recent studies, they have also employed the grey literature to identify five testing practices for IaC scripts [25] and ten Kubernetes-specific security practices [24]. Guerriero et al. [12] identified four best practices and seven bad practices from their survey with practitioners.

3.1.4. Previous Literature Reviews concerning IaC-related topics

Previous grey and white literature surveys analyzed topics related to Infrastructure as Code such as DevOps tools [36], cloud resource orchestration techniques [37], and cloud deployment modeling tools [38]. These surveys describe the general capabilities of IaC tools and their usage scenarios. However, there is little or no information about the best and bad practices of using such tools.

3.2. Research goals

Given the work above, it is clear that best and bad practices regarding IaC should be derived from grey literature. Therefore, our goal is to analyze, assess, and summarize such literature sources systematically. We aim at compiling a *comprehensive* catalog of best and bad practices that developers should follow when developing and maintaining IaC projects. Please note that our scope is not limited to source code but compass the analysis of the different aspects of IaC projects by reporting the practices for three widely used languages, namely

Ansible, Chef, and Puppet. Finally, we aim to define and characterize the concept of IaC comprehensively.

4. Research Methodology

We build our research design upon the SLR guidelines proposed in systematic literature reviews in software engineering [39, 40]. We also used the recent grey and multi-vocal SLRs [41, 42, 43, 44] as a reference. Figure 2 shows the steps of our SGLR process.

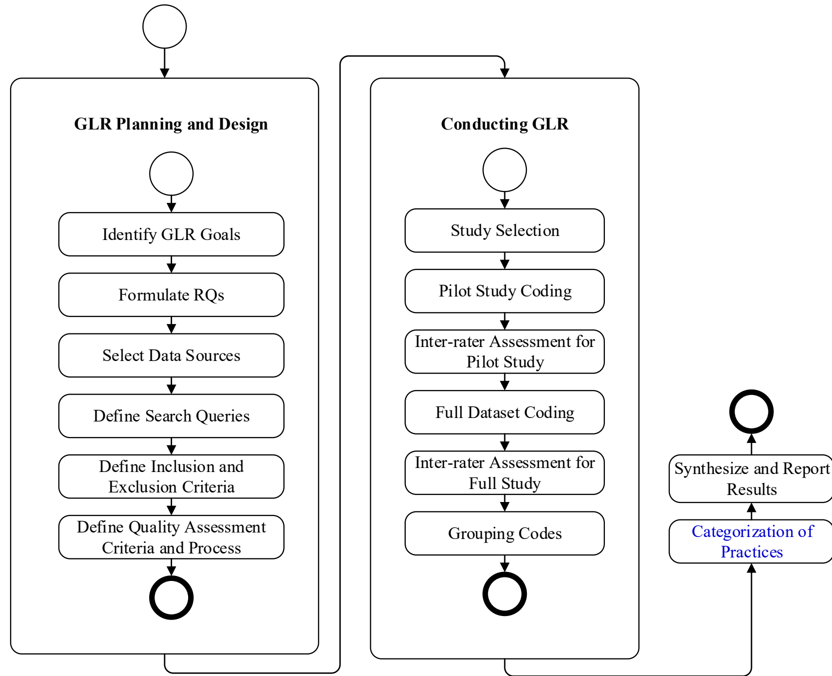

Figure 2: An overview of our SGLR process

4.1. Research Questions

We defined three research questions to achieve our research goals, as mentioned in Section 3.2.

- **RQ1: How is IaC defined and classified by developers?** RQ1 aims to understand and characterize the concept of IaC from the standpoint of

a practitioner. There exist different categories of IaC, and practitioners use several IaC tools, which exhibit common and different functional capabilities. RQ1 thus defines and classifies IaC and identifies the functional features of IaC.

- **RQ2: Which are the best practices adopted by developers in developing IaC?** By adhering to IaC best/good practices, practitioners can create IaC artifacts with the desired quality. There exist different IaC languages, and each language includes common and differing best practices with others. Hence, RQ2 attempts to compile a unified catalog of IaC best practices.

- **RQ3: Which are the bad practices adopted by developers in developing IaC?** By following bad practices, practitioners inadvertently can introduce smells into IaC artifacts. As the bad IaC development practices vary across different IaC languages, while sharing some commonalities, RQ3 aims to formulate a unified catalog of IaC bad practices.

4.2. Data Sources and Search Strategy

Similar to other multi-vocal and grey literature reviews [41, 42, 43, 44], we employed the Google search engine to search the grey literature. We only consider textual sources such as reports, blog posts, white papers, and official documentation of each IaC language. We first identified the initial set of search terms on the research questions and created the following query:

```
Infrastructure as code (bug(s)|defect(s)|fault(s)|(anti-)pattern(s))
```

We obtained only a few sources; thus, we refined the generic terms of the query to include the language-specific keywords. In this study, we only considered three IaC languages such as Ansible, Puppet, and Chef. Two main reasons are driving this choice. On the one hand, these are the most popular IaC languages in the industry according to our previous survey [12]. On the other hand, the inclusion of a larger number of smaller and less popular languages would have resulted in a prohibitively expensive manual screening.

This resulted in the following query:

```
(ansible|puppet|chef) ((anti-)pattern(s)|(best|good|bad) practices)
```

We applied the above two queries on the Google search engine, scanning each resulting page until saturation (i.e., we stopped our search when no new relevant articles were emerging from search results) (as in [41, 44]). We performed our search in an incognito-mode so that we avoided our personal search bias.

Table 1: Queries and number of articles found and included.

#	Query	total	Incl. 1st round
1	Infrastructure as code	24	13
2	Infrastructure as code bugs	31	13
3	Infrastructure as code defects	27	10
4	Infrastructure as code anti-patterns	27	9
5	puppet anti patterns	17	5
6	puppet best practices	19	7
7	chef anti patterns	15	8
8	chef best practices	23	10
9	ansible anti patterns	19	6
10	ansible best practices	18	11
	Total	220	92
	After I/E criteria		-25
			67

Table 1 summarizes the potentially relevant results for each search string in Column *total* (220 sources overall), after an initial pre-filtering of e.g., articles not related to the topic, Wikipedia entries, repeated entries, or paid books. Next, to select the most relevant candidate articles, we applied the Inclusion/Exclusion Criteria described in Section 4.3, by reading through the main sources. With those criteria we filtered out 128 sources, for a final list of 92 relevant sources (Column *include* in Table 1).

4.3. Eligibility Criteria and Study Selection

Inclusion/Exclusion Criteria. We considered an article for further analysis only when it satisfies all inclusion criteria and does not satisfy any of the exclusion criteria. We included:

- Articles in English and the full text is accessible;
- Articles matched the focus of the study, i.e., concepts and characteristics, best/good and bad practices, bugs/defects/anti-patterns, and challenges concerning IaC in general or a specific IaC language.

We excluded:

- Articles not matching the focus of the study;
- Articles restricted with a paywall;
- All duplicate articles found from various sources;
- Short elements that do not contain sufficient data for our study, such as posts in forums and comments;
- Articles that do not provide scope, consequences, and examples of the proposed best/bad practices and bugs/defects/anti-patterns.

Quality assessment. In addition to the aforementioned inclusion/exclusion criteria, we applied the following criteria for further assessing the quality of the articles, which were adopted from the existing grey and multi-vocal literature review studies [41, 45]:

- Is the publishing organization reputable?
- Is an individual author associated with a reputable organization?
- Has the author published other work in the field?
- Does the author have expertise in the area?
- Does the source have a clearly stated purpose?
- Is the source recent enough (i.e., within the last three years)?

The validation was mainly carried out by two of the authors of this paper. The authors distributed the material between them nearly equally, and they

validated only their corresponding instances. In problematic cases, the authors mutually agreed on whether those specific documents should be considered. Whenever they did not agree, they discussed with one or more other authors of the paper to resolve the disagreement. Initially, we considered only the sources that reached at least 3 of the above criteria, for a total of 50 sources. However, we could not answer some of the criteria/questions for the rest of the sources. For example, in some cases, there was no indication of the publishing organization or author. From the remaining sources, we selected another 15 relevant (given that they covered the topic of interest), albeit not covering those criteria. All in all, we proceeded with a final set of 67 selected sources.

4.4. Data Synthesis and Analysis

To attain the results for answering our research questions, we read, synthesized, and analyzed the above selected industrial studies following a qualitative analysis process [46]. In particular, to obtain codes, groupings, and categories related to the research questions systematically, we followed a series of steps:

1. **Pilot study**. The first set of 20 sources were randomly selected and analyzed independently by two researchers to establish an initial set of codes, using Structural and Descriptive Coding [47]. The codes were extracted by conceptualizing all the information stemming from the sources related to the Research Questions. This phase included a constant back and forth check of the codes, constantly refining them to sharpen the growing theory.
2. **Inter-rater assessment (pilot study)**. After the coding phase, the two researchers started an inter-rater assessment on the codes to appraise each other's codes and reach unanimity on their names, types, and categories. This process led to the change of several codes, reaching uniformity and rigidness among them.
3. **Full dataset coding**. The rest of the sources were coded next, following the consensus reached by the two researchers in the previous step. The

sources were split into two halves, each analyzed and coded by one of the researchers independently.

4. **Inter-rater assessment (full study)**. After the independent code extraction, the researchers looked into each other's codes to inter-rater assess the work and make sure everything was coded, as previously stated. All the discrepancies were solved via discussions.
5. **Grouping**. The theory led us towards six overarching groups of codes: best practices, bad practices, IaC definition, IaC advantages, and IaC challenges. These groups were then analyzed and further decomposed as necessary. The best/bad practices were grouped into language-agnostic, Ansible, Chef, and Puppet.
6. **Categorization of Practices**. We categorized the identified atomic best and bad practices into coarse-grained categories. For the candidate practice categories, we used the practice categories proposed in the existing literature [48, 49, 50, 51, 52, 53]. We also employed the categories proposed in the Common Weakness Enumeration (CWE)[1], which was used by several studies on smell and bug taxonomies [54, 33]. The first authors of this paper first compiled a list of candidate categories. Next, the first author and the fourth author independently assigned the atomic practices into a subset of the candidate categories. We addressed all the discrepancies through discussions. When needed, we modified some categories in the selected candidate categories to reflect the IaC context.

4.5. Replication Package

To enable further validation and replication of our study, we made the related data available online [2]. It contains the full list of sources, the qualitative analysis (codes, groupings, and analysis) performed with the *Atlas.ti* tool[3], the keywords

[1]https://cwe.mitre.org/
[2]https://github.com/IndikaKuma/ISTGrayIAC
[3]https://atlasti.com/

and phrases extracted to define IaC in RQ1, the atomic best/bad practices, and the final taxonomy derived for RQ2 and RQ3.

5. Analysis of the Results

In this section, we provide an overview of the selected grey literature, and answer each research question in detail.

5.1. Overview of the Selected Grey Literature

Table 2 shows provide an overview of the selected studies, including contribution type, content type, IaC language, publication year, and publication venue. The sources cover the articles from IT companies with high reputation (e.g., Microsoft and IBM), official communication channels of the IaC languages/tools (Ansible, Chef, Puppet), articles on online publishing platforms (e.g., DZone and Medium), and blogs.

Table 2: Overview of the selected industrial studies

Study	Type	Content	Lang.	Year	Venue
[S1]	Doc	Def	-	2017	Microsoft
[S2]	Blog	Def/Features	-	2018	Thorntech
[S3]	Blog	Def/Features	-	2019	TechTarget
[S4]	Blog	Def/Features Adv/Challenges	-	2016	ThoughtWorks
[S5]	Tutorial	Features/Adv	Ansible	2018	Hacker Noon
[S6]	Blog	Def/Adv/Best Bad/Features	-	2018	BMC Software
[S7]	Article	Def/Best Features	-	2017	TechBeacon
[S8]	Blog	Def/Features Challenges	-	2018	CloudBees
[S9]	Article	Best	-	2018	DZone
[S10]	Blog	Def/Features	-	2018	IBM
[S11]	Blog	Features/Adv	-	2018	IBM
[S12]	Blog	Def/Features/Adv Best/Challenges	-	2018	Rackspace
[S13]	Blog	Def/Features Best	-	2018	Qualibrate

[S14]	Blog	Def/Best/Adv Features	-	2018	IbexLabs
[S15]	Blog	Def/Adv	-	2018	Red Badger
[S16]	Blog	Def/Adv	-	2018	Thorntech
[S17]	Blog	Best/Challenges	Puppet	2016	Puppet
[S18]	Blog	Best/Features Def/Challenges	-	2018	OVO Energy
[S19]	Article	Best/Adv Features	-	2018	Medium
[S20]	Blog	Best/Def Challenges	Puppet	2018	ServerlessOps
[S21]	Blog	Def/Challenges	-	2018	The New Stack
[S22]	Blog	Best/Feature	-	2018	Thorn Technologies
[S23]	Article	Def/Adv	-	2019	Medium
[S24]	White Paper	Def/Features/Adv	-	2018	Risk Focus
[S25]	Blog	Def/Features Adv	-	2017	Tasktop Technologies
[S26]	Blog	Def/Feature/Best Adv/Challenges	-	2018	Network Computing
[S27]	Blog	Best	-	2018	OpenCredo
[S28]	Slides	Def/Bad/Best	-	2015	SlideShare
[S29]	Article	Bad	-	2018	DZone
[S30]	Blog	Best/Bad	-	2017	Hacker Noon
[S31]	Blog	Bad/Best	-	-	Qualysoft
[S32]	Blog	Features/Def Best	-	2018	Pythian Group
[S33]	Slides	Bad	Puppet	2017	GitHub
[S34]	Blog	Bad	Puppet	2016	pysysops
[S35]	Blog	Best	Puppet	2016	blog.danzil.io
[S36]	Blog	Best	Puppet	2014	garylarizza
[S37]	Slides	Best/Bad	Puppet	2016	SlideShare
[S38]	Doc	Best/Bad	Puppet	2018	GitHub
[S39]	Slides	Best	Puppet	2016	SlideShare
[S40]	Blog	Best/Bad	Chef	2017	Chef
[S41]	Doc	Best/Bad	Chef	2015	GitHub
[S42]	Blog	Best/Bad	Chef	2015	LinkedIn
[S43]	Slides	Best/Bad	Chef	2015	SlideShare
[S44]	Blog	Best	Chef	2015	Ragnarson
[S45]	Article	Best/Challenges	Chef	2017	Medium
[S46]	Doc	Best	Ansible	2019	Ansible
[S47]	Blog	Bad	Ansible	2015	GitHub
[S48]	Blog	Best/Features	Ansible	2016	Ansible
[S49]	Blog	Best	Ansible	2018	New Relic Software

[S50]	Slides	Best/Bad	Ansible	2017	SlideShare
[S51]	Article	Best/Bad	Ansible	2018	serverraumgeschichten
[S52]	Blog	Best	Ansible	2018	Wordpress
[S53]	Tutorial	Best	Ansible	2017	GitHub
[S54]	Blog	Best/Features	Ansible	2018	juliosblog
[S55]	Article	Best	Ansible	2018	InfinityPP.com
[S56]	Blog	Best	Ansible	2017	OzNetNerd
[S57]	Blog	Best/Bad	Ansible	2017	Andreas Sommer
[S58]	Blog	Best/Bad	Puppet	2017	Puppet
[S59]	Blog	Adv/Best Features	-	2014	DevOps
[S60]	Slides	Def/Feature Challenges/Best	-	2013	OWASP
[S61]	Blog	Best/Features	Puppet	2015	DevOps
[S62]	Blog	Best	Puppet	2013	Radiant3 Productions
[S63]	Blog	Best/Bad	Chef	2015	Chef
[S64]	Blog	Best	Puppet	2013	glennposton.com
[S65]	Blog	Bad	Ansible	2018	oteemo.com
[S66]	Doc	Best/Bad	Chef	2019	Chef.io
[S67]	Doc	Best/Bad	Puppet	2019	Puppet.com

Regarding the overall reach and acceptance of the considered literature, unfortunately, most of the venues do not offer comments or likes to see if there is a reaction and whether it is positive or negative. From the few that do so, the most commented and liked publications come from the blogging platforms `Hackernoon.com` (5 comments and 500 likes/claps on average) and `Medium.com` (40 likes/claps). Conclusively, we see an increasing trend in the quality and quantity of IaC grey literature, confirmed by diversification of publication venues, authors, and their companies.

Figure 3 illustrates the distribution of the selected studies by the contribution type, specific IaC languages, and key concepts. The blogs followed by the articles on the online publishing platforms are the predominant publishing media for the practitioners. The results indicate that the practitioners discuss IaC in general as well as specific IaC technologies. As IaC is a relatively new concept, more attention has been paid to understand and characterize IaC. The best and bad practices that help to develop high-quality IaC have also been a key focus.

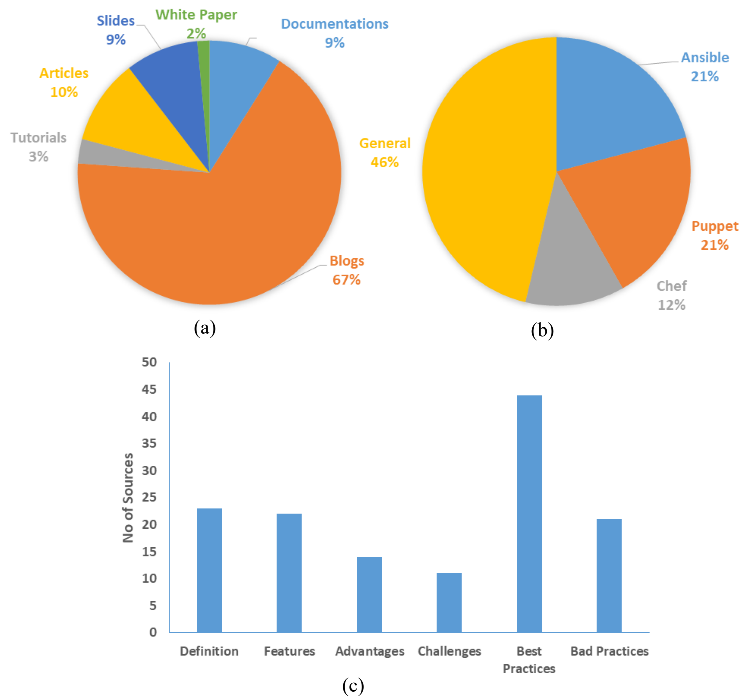

Figure 3: Distribution of selected studies by (a) by contribution type, (b) IaC language, (c) content focus

Figure 4 shows the outcome of the frequency analysis of the open codes from the qualitative analysis of the selected literature, highlighting the importance of the key topics (codes). In general, the IaC patterns have been discussed frequently as its counterpart. The selected literature has also reasonably considered the characteristics, advantages, and challenges of IaC.

5.2. RQ1: Definition Proposal for IaC

In this section, we use the definitions, classifications, and features of IaC found in the selected studies to provide an integrated definition for IaC. We gathered the definitions of IaC from six different sources in our grey literature and extracted the keywords and concepts from the corresponding text by manually scanning the sources. The extracted data is available in the replication package of this study. We identified three major dimensions to explain IaC: types of management

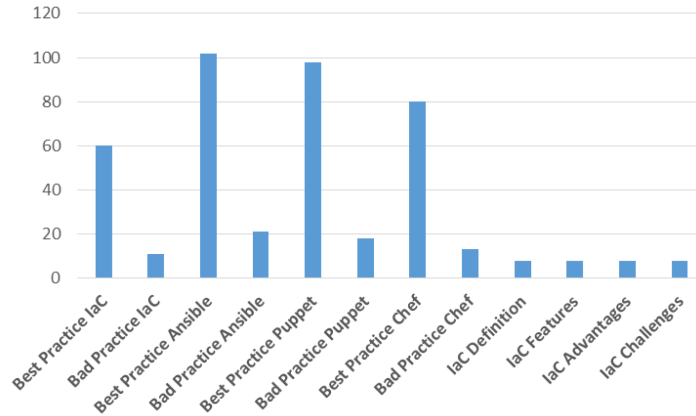

Figure 4: Distribution of code groupings stemming from qualitative analysis

operations (on a computing environment) supported by IaC, principles, and methods of realizing such management operations with IaC, and the desired properties of the managed environment. The number of the extracted keywords and clauses for the three dimensions are 46, 35, and 26, respectively.

5.2.1. Management Tasks

IaC supports the management of the entire lifecycle of a computing environment consisting of infrastructure, software/platform, and applications.

- *Infrastructure* includes the fundamental computing resources such as servers, networks, and storage. The management capabilities include: (1) definition of the desired state of the infrastructure, (2) provisioning of the infrastructure by enacting its definition, (3) versioning the infrastructure by versioning its definition, (4) switching between different infrastructure variants by switching between their definitions, (5) cloning infrastructure variants by cloning their definitions, (6) destroying the infrastructure variants based on their definitions. Versioning of infrastructure includes adding/removing/updating individual computing resources and updating their composition topology.
- *Software/Platforms* are used to deploy, run, and manage applications,

such as programming languages, frameworks, libraries, services, and tools. IaC supports (1) defining the desired state of the software/platform (e.g., MySQL is installed with the root user), (2) installing, (re)configuring, and uninstalling the software/platform based on its definition.

- *Applications*-specific capabilities are (1) defining the desired state of the application deployment, (2) deploying, (re)configuring, un-deploying the application using its deployment definition.

In the selected literature, the IaC tasks are broadly divided into: infrastructure templating, configuration orchestration, configuration management. The infrastructure templating generates images of the infrastructure resources such as virtual machines and containers. The configuration orchestration provides infrastructure resources, while the configuration management installs software/-platforms/applications on the provisioned infrastructure and manages their configurations. The management tasks performed through IaC exhibit the two key properties:

- *Idempotence* of a task makes the multiple executions of it yielding the same result. The repeatable tasks make the overall automation process robust and iterative (i.e., the environment can be converted to the desired state in multiple iterations.)

- *Transparency* of a task is achieved by making the state of the environment explicit and visible (via code) to the teams managing the environment. IaC code represents the infrastructure's documentation, and IaC files are versioned controlled, providing a clear record of changes made to the environment.

5.2.2. Methods

IaC replaces the conventional processes used to managing a computing environment with a process that enables applying software engineering practices. Instead of low-level shell scripting languages, the IaC process uses high-level

domain-specific languages that can be used to design, build, and test the computing environment as if it is a software application/project. The conventional management tools such as interactive shells and UI consoles are replaced by the tools that can generate an entire environment based on a descriptive model of the environment.

There are two main programming models for IaC languages: declarative and imperative (procedural). In the declarative model, the developers define the desired end state of the environment and let IaC tools determine how to achieve the defined state. In the imperative model, the developers need to specify the process that transforms the current state of the environment to the desired end state as an ordered set of steps. Puppet uses a declarative style, whereas Chef and Ansible use an imperative style.

Any software engineering methodology can be used for building and managing IaC projects. However, on the one hand, most practitioners advocate adopting DevOps practices to ensure IaC source code is developed, tested, versioned, and updated continuously by operation and development teams in collaboration. On the other hand, IaC is considered a critical enabling technology for DevOps and CI/CD. IaC supports the steps of a CI/CD pipeline, such as packaging (application) and configuring (the environment, including the application). The applications can be tested and validated using dynamically provisioned test environments consistent with the production environment.

5.2.3. Properties of Managed Environments

From the selected literature, we can find the environment properties that enable IaC and those that IaC induces. Figure 5 depicts the frequency of codes for those properties.

- *Virtualization* enables on-demand provisioning of fundamental computing resources such as virtual machines and containers and is thus considered a prerequisite for IaC. Virtualization provides an additional layer of abstraction from provisioning and configuration.

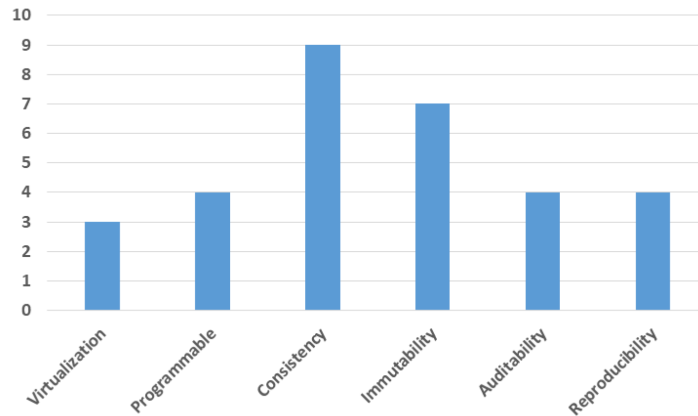

Figure 5: Frequency of codes for the key properties for an environment managed by IaC

- *Software-defined/Programmable* infrastructure consists of abstracted and virtualized computing resources and offers APIs to create and manage those resources. It is also a prerequisite for IaC since IaC needs to provide and configure infrastructure resources on-demand pragmatically. On the other hand, IaC also makes the systems and processes employed to manage the infrastructure *software-defined* as their management functions are now codified and exposed as APIs.

- *Consistency* among multiple environments (development, test, production) can be achieved using IaC. Indeed, IaC eliminates the so-called *environment/configuration drift*, a phenomenon that occurs when configuring multiple deployment environments. It leads to unique, irreproducible configurations that are due to uncoordinated changes over time. With IaC, the changes are performed to the IaC source code (environment state definition files) that is properly versioned controlled.

- *Immutability* of infrastructure indicates that the infrastructure cannot be modified once it is provisioned. The only way to change the infrastructure is to replace it with a new version. Immutable infrastructures eliminate configuration drift and thus simplify maintaining the consistency between

different environments. IaC makes immutable infrastructure feasible and practical as a new infrastructure can be created, versioned, and destroyed quickly with IaC.

- *Auditability* of the computing environment is the ability to track and trace the changes to the environment. As the changes to the environment are performed by changing the corresponding IaC source code, a version-controlled IaC provides a detailed audit trail for changes.

- *Reproducibility* of the environment indicates the degree to which a given environment can be easily, rapidly, and consistently recreated. With IaC, a given version of an environment can be provisioned using the same version of the environment definition model stored in the source code repository.

5.2.4. An Integrated Definition of IaC

Infrastructure-as-Code (IaC) is a model for provisioning and managing a computing environment using the explicit definition of the desired state of the environment in source code and applying software engineering principles, methodologies, and tools. IaC DSLs enable defining the environment state as a software program, and IaC tools enable managing the environment based on such programs. The managed computing environment comprises three key types of computing resources (i.e., infrastructure, software/platform, and application) and exhibits six essential properties (i.e., Virtualized, Software-defined/Programmable, Immutable, Auditable, Consistent, and Reproducible). The management of the environment encompasses the management of the lifecycles of each computing resource in the management.

IaC Definition

The term IaC can be comprehensively analyzed and described using three major dimensions: (1) the three types of management operations supported by IaC, (2) the methods for implementing such management operations with IaC, and (3) the six desired properties of the managed environment.

5.3. RQ2: IaC Best Practices

In this section, we present IaC best practices extracted from the selected literature through the mixed-methods approach. Table 3 shows the categories of the IaC best practices described in the selected studies, extracted using the qualitative analysis. Each best practice category consists of several sub-categories. For each sub-category, the sources, the number of related codes (frequency of codes) in sources, and the number of atomic practices are also shown. The code frequency of a sub-category is constituted by adding the code frequency of its atomic practices.

Table 3: Best practice categories and their sources (#), code frequencies (C), and number of atomic practices (P)

Practice	Independent			Ansible			Chef			Puppet		
	#	P	C	#	P	C	#	P	C	#	P	C
1 Write IaC programs for people, not computers												
1a Make names consistent, distinctive, and meaningful	-	-	-	S46 S48 S50 S51 S52 S53 S54 S55 S57	5	22	S41 S43 S66	7	8	S38 S39 S67	8	9
1b Make code style and formatting consistent	-	-	-	S46 S48 S50 S52 S54 S55 S57	2	10	S41 S45 S66	2	4	S67	5	5
1c Make parameters, their types, and defaults explicit	-	-	-	S46 S51 S54	2	4	-	-	-	S38 S67	4	5
1d Use conditionals properly	-	-	-	-	-	-	S41 S44 S63	2	3	S67	3	3
2 Do not repeat yourself (or others)												

2a Modularize IaC programs	S08 S18 S22 S28	1	5	S46 S47 S49 S55 S57	8	17	S40 S41 S43 S44	5	6	S20 S37 S38 S39 S67	12	15
2b Re-use code instead of rewriting it	S19 S28	3	4	S49 S50 S51 S54 S55	2	6	S40 S41 S42 S44 S63 S66	10	15	S37 S38 S39	9	11
2c Select the right modules for the job and use it correctly	-	-	-	S48 S49 S51 S55	2	9	-	-	-	S38 S67	2	2
2d Reuse the tools that the community use	-	-	-	-	-	-	S41 S42 S44	2	3	S38 S61 S62	3	7
3 Let the IaC tools do the work												
3a Codify everything	S08 S09 S14 S22	1	4	-	-	-	-	-	-	-	-	-
3b Package applications for deployment	S28	1	1	-	-	-	-	-	-	-	-	-
3c Do not violate immutability and reproducibility of your infrastructure	S10 S21 S22 S28 S31	2	6	-	-	-	-	-	-	-	-	-
3d Do not violate idempotence of IaC programs	-	-	-	S49 S57	2	3	S41 S63	2	3	S38	2	2
4 Make incremental changes												

4a Use a version control system	S02 S04 S06 S07 S08 S13 S14 S19 S22 S26 S27 S28 S29 S32 S33 S61	2	17	S46	1	1	S41	3	3	S67	1	1
4b Favor versionable functionalities	-	-	-	-	-	-	S40 S42 S44	2	3	-	-	-
5 Prevent avoidable mistakes												
5a Use the correct quoting style	-	-	-	S50 S57	1	2	S41 S66	2	2	S67	2	2
5b Avoid unexpected behaviors whenever possible				S54 S57	2	2	S41	2	2	S67	2	2
5c Use proper values				S46	1	1	S66	4	4	S38 S67	1	2
6 Plan for unavoidable mistakes												
6a Write tests as you code	S06 S07 S09 S17 S22 S27 S28 S61	1	12	S50 S54 S57	1	3	S41	2	2	S38 S61 S62	2	3
6b Do not ignore errors	-	-	-	S57	1	1	-	-	-	-	-	-
6c Use off-the-shelf testing libraries	-	-	-	-	-	-	S41	2	2	-	-	-
6d Monitors your environment	S28	2	2	S47	1	1	-	-	-	S38	1	1
7 Document little but well												

7a Code as documentation	S07 S09 S20 S22 S28	1	7	-	-	-	-	-	-	-	-	-
7b Use document templates	-	-	-	S50 S51	1	2	-	-	-	S67	2	2
8 Organize repositories well												
8a Modularize repositories	S32	1	1	-	-	-	S41	1	1	S38	3	3
8b Use standard folder structures	-	-	-	S46 S51 S54 S57	2	5	S41	2	2	S38 S67	2	3
9 Separate configuration data from code												
9a Use configuration datasource	S28	1	1	-	-	-	S41	2	2	S38 S58 S67	7	9
9b Modularize configuration data	-	-	-	S46 S57 S55	2	5	-	-	-	S37 S38	1	2
9c Select data sources wisely	-	-	-	-	-	-	S41 S43	6	8	-	-	-
9d Use configuration templates	S28	1	1	S49 S55	1	3	S41	2	2	S38 S51 S67	2	3
10 Secure configuration data and write secure code												
10a Isolate secrets from code	S28	1	3	S51 S53 S55	1	3	S41 S63	2	3	S61	1	2
10b Protect your data at rest	S28	1	1	-	-	-	S41	2	2	-	-	-
10c Use facts from trusted sources	-	-	-	-	-	-	-	-	-	S38 S67	2	3
10d Use standard secure coding practices	-	-	-	S57	2	2	-	-	-	S67	2	2

5.3.1. Practice 1 - Write IaC programs for people, not computers

This category includes best practices affecting readability, understandability, and maintainability of IaC programs.

Make names consistent, distinctive, and meaningful (1a). Ansible practices provide guidelines concerning the naming of tasks and variables. Each task should have a name that better communicates the purpose of the task to the users. As the task names are visible in the task execution's output, they can be enriched with the variables to help debug tasks. The names of the variables can include usage context, such as role name, to help the users quickly figure out the origin of a variable. The use of a single naming style for variables also improves the readability. In Chef, the name of an application cookbook should include its usage contexts, such as application name and organization name. Nodes, attributes, recipes, environments, and roles can be configured using appropriate naming conventions when defining their attributes. Name collisions between attributes defined in different cookbooks can be avoided by indicating the name of owning cookbook in the data bag name. Furthermore, environment names should always be in uppercase. Puppet practitioners also recommend using descriptive names that reflect their purpose for Puppet constructs such as profiles, roles, modules, resource types, and variables. More constraints are defined for the variables, the legality of the characters used, and the use of qualified names when referring to them.

Make code style and formatting consistent (1b). For each IaC language, the proper order of information with different language constructs is recommended. For example, attribute ordering in a resource declaration and ordering of resources and parameters with a class (Puppet), and context organization of Ansible playbooks and inventories. Ansible uses YAML syntax. Thus, IaC developers need to follow the best practices of YAML, particularly syntax and conventions (e.g., starting a script with three dashes, two spaces indentation, and using the map syntax). Regarding Puppet, the formatting best practices provide guidelines on spacing, indentation, and formatting complex types such as arrays and hashes to enhance code readability.

Make parameters, their types, and defaults explicit (1c). Each IaC language has constructs to define and configure parameters (e.g., Ansible tasks, Chef resources,

and Puppet classes). The explicit declaration of all necessary parameters, including their types and defaults, makes their logic readable and understandable and can reduce errors due to incorrect assumptions concerning behaviors and parameters.

Use conditionals properly (1d). The improper use of conditionals can make the code complex and hamper the readability and understandability of codes. When using Puppet conditionals, there are recommendations to minimize these adverse effects: separation of conditionals on data assignment from resource declaration, explicit default clauses, correct alignment of if/else statements, and simple short logical expressions.

5.3.2. Practice 2 - Do not repeat yourself (or others)

This category relates to the IaC best practices that aim to reduce code clones and increase the reuse of IaC programs and tools.

Modularize IaC programs (2a). The IaC programs should be decomposed into modular fragments and stored in a repository to be dynamically discovered and composed as necessary. This decomposition enables greater control over who has access to which parts of the infrastructure code and reduces the number of changes needed for infrastructure configurations. Each IaC language provides mechanisms to develop module IaC programs.

In Ansible, playbooks can be modularized based on their purposes, such as web servers and database servers. The top-level (master) playbook refers to the role-based children playbooks. This practice allows for selectively configuring the infrastructure. Ansible supports both infrastructure configuration and application deployment: different playbooks and roles can separate the two concepts. Furthermore, the parameterized role pattern can be used to provide configurable (site-specific) behaviors. The default values for the role parameters can be overridden during the instantiation of the role. To select and execute a subset of tasks in a play, Ansible supports the concept of tags, which are task-level attributes. While the tags can ease debugging and testing of a play,

they reduce readability and maintainability as the control flow of a play becomes complex and hidden and can cause the erroneous execution of plays. Hence, decomposing complex playbooks into a set of smaller task-focused playbooks is preferred over using tags.

In Chef, both cookbooks and recipes can be modularized and parameterized. A recipe defines a collection of resources to be created and managed on a node and provides a public API for cookbooks. Recipes should be created focusing on related sets of tasks, such as deploying and configuring a specific application component.

Puppet profiles and roles help build reusable and customizable system configurations by separating component modules into class assignments and nodes using a two-layer indirection. Profiles use fine-grained reusable component modules to deploy and configure a logical technical stack, for example, installing and configuring Apache and deploying and configuring a PHP web application. Roles use profiles to build a complete system configuration or define the type of server. For example, a web server includes profiles for installing prerequisites, Apache, and PHP application deployment. Roles are assigned to nodes (so-called node classification). To maintain the modularity of Puppet modules, developers can (i) create different classes for different concerns (e.g., installing software and configuring the installed software), (ii) hide the private behaviors of the modules with private classes, (iii) adhere to the single responsibility principle, and (iv) split the content into files. Similarly, to improve class modularity, developers can (i) split class files, (ii) parameterize classes, and (iii) define public and private classes.

Reuse code instead of rewriting it (2b). IaC languages support the reuse of IaC code units. The Wrapper pattern can be used to extend and customize Ansible roles to be shared in Ansible Galaxy. Similarly, Puppet resources and tasks can be shared in Puppet Forge, and Chef cookbooks can be shared in Chef Supermarket applying the Repository pattern. IaC languages also enable writing custom codes that are reusable across different computing platforms by

encapsulating and simplifying the logic for managing dependencies and accessing platforms.

Select the right modules for the job and use it correctly (2c). Each IaC language enables reusing infrastructure management functionalities as modules, for example, Ansible modules and roles, Chef resources and cookbooks, and Puppet modules. Practitioners need to select the appropriate module for a task and then correctly use and configure it. For example, there are thousands of Ansible modules with specific focuses (e.g., install a service, download a file, and unzip a file). Many modules are operating system agnostic, idempotent, and provide high-level abstractions to ease the automated management of different environments. Thus, task-specific modules are preferred over the general modules that execute ad-hoc OS-level commands (e.g., shell and command modules). There are also recommendations for the usage of specific Ansible modules. For example, managing the configuration files as whole with template modules is preferred over making uncoordinated fine-grained changes to the configuration files with the lineinfile or blockinfile modules. The correct usage of the service module is to reload the updated configuration settings of system services gracefully without making the services inactive.

Reuse the tools that the community use (2d). IaC practitioners recommend using the tools and deployment patterns developed by the respective communities. For example, Chef recommends specific tools and systems for testing and sharing the cookbooks, generating data bags, and managing dependencies between cookbooks. Similarly, there are deployment architectures and development tools approved by the Ansible and Puppet communities.

5.3.3. Practice 3 - Let the IaC tools do the work

IaC enables seamless deployment of the infrastructure solely based on the configuration and minimizes or eliminates manual post adjustments. This category consists of the practices that allow for gaining all benefits and capabilities of the IaC model.

Codify everything (3a). The practitioners recommend codifying everything needed to provide and manage a computing environment, treat IaC code as the documentation for managing an environment, and avoid putting additional instructions in a separate document. All infrastructure specifications should be explicitly coded in the configuration files, which act as the single truth source of the infrastructure specifications.

Package applications for deployment (3b). Furthermore, an application needs to be packaged to ease its deployment and execution, for example, *.war* file for a web application or a Docker image for a web server. Packaging can reduce the amount of code needed in later stages of configuration management.

Do not violate immutability and reproducibility of your infrastructure (3c). As discussed in Section 5.2, infrastructure immutability simplifies management and improves predictability. The *Environment Template* (e.g., Docker and Docker compose files) simplifies cloning, sharing, and versioning environments.

Do not violate idempotence of IaC programs (3d). Practices defying the idempotency of IaC programs are not recommended; for example, skipping tasks in Ansible plays, using custom imperative resource management code – ruby blocks and bash commands in Chef recipes. To achieve idempotence, Chef provides mechanisms called guard properties that can be used to decide if executing or not a resource based on the current state of the node. In Puppet, Lightweight resource providers (LWRP) enable creating custom resources by extending the existing other resources if necessary. When writing an LWRP, it is recommended to define the actions for managing the complete lifecycle of the custom resources necessary to maintain idempotence.

5.3.4. Practice 4 - Make incremental changes

This category relates to the practices that aim to ensure managed changes to the infrastructure configurations.

Use a version control system (4a). With IaC, all configuration details are written in code using a version control system: any changes to the infrastructure configuration can be managed, tracked, and reconciled. The version control system provides an audit trail for code changes, the ability to collaborate, peer-review IaC code, and easily define and clone configurations and make changes seamlessly and consistently. Semantic versioning is recommended for each IaC language. For Chef, additional guidelines were provided. In this language, cookbooks are the main versioned artifact. The optimistic version constraints in their dependencies make the adoption of newly developed cookbooks easy. Given that the highest-numbered cookbook is always used, the version of a cookbook should not be decreased. It is recommended to store every configuration, documentation, test case, and script under version control. Binary files such as large VM images or Docker images should be stored in separate locations.

Favor versionable functionalities (4b). As the changes to the non-versionable functionalities have global effects, the versionable alternatives are preferred. In Chef, a Role is a logical way to group nodes (e.g., web servers and databases). Each role can include zero (or more) attributes and a run-list of recipes. Roles are not versionable. Therefore, they should not be used to keep the recipes run-lists, which should be assigned as default recipes of cookbooks.

5.3.5. Practice 5 - Prevent avoidable mistakes

This category includes the practices that can be used to prevent introducing faulty behaviors to IaC codes.

Use the correct quoting style (5a). The improper usage of quotes can also introduce undesired errors such as erroneous interpolation of values. Thus each IaC language provides the best practices for using single quotes and double-quotes.

Avoid unexpected behaviors whenever possible (5b). Some best practices aim at preventing the introduction of unexpected problematic behaviors. Persistent node attributes are not explicit in Chef, leading to unexpected behaviors;

thus, they should be avoided. Instead, Roles or environments can contain the attributes shared by several cookbooks. External attributes make managing dependencies and refactoring cookbooks problematic. Hence, minimizing their usage and making them visible are recommended. Using separate attribute files for capturing those attributes, which recipes override, makes explicit the attributes' purpose. This solution reduces the undesired errors that can occur due to the unknown execution orders of recipes at design time. In these cases, changes to attribute values caused by overriding attributes will lead to erroneous or inconsistent values. For this reason, arrays are not recommended to define composite attributes as merging their attributes is trickier than merging hashes. The misuse of variables and their precedence is a common cause of unexpected errors. Variables should be defined a close as possible to their usage, for example, Ansible role variables under roles and host variables in inventory files. *group_vars* and *host_vars* enable scoping variables to particular groups or systems.

Use proper values (5c). Each IaC languages have constraints on the values allowed for specific variables/parameters/properties (e.g., file modes and file paths). For example, a valid 4-digit octal value or symbol should be used for file modes.

5.3.6. Practice 6 - Plan for unavoidable mistakes

Mistakes are inevitable; therefore, some practices aim at verifying and maintaining the validity of code over time.

Write tests as you code (6a). Standard testing practices such as static code analysis with Lint tools, unit testing, integration testing, functional testing, and test automation should also be applied to IaC testing. Lint tools can be used to detect syntactical/structural errors and bad coding practices. Unit testing can validate the units of infrastructure codes (e.g., downloading software, installing software, enforcing a password policy, and configuring a firewall). Integration testing can validate the collective behavior of multiple IaC units (e.g., deploying a 3-tier web application). Functional testing can test higher-level assertions

concerning installed applications and services (e.g., check if a system user can log in to the environment and perform some operations). Automated tests can be configured to allows for continuous testing of configuration code. The changes to production servers can be tested with dry-run redeployments. As the developers identify issues, they can add new test cases to the test suites to be automatically invoked.

Do not ignore errors (6b). The errors during task execution should not be ignored but handled with the error handling capabilities of Ansible to ensure that the execution of a task leaves the node/infrastructure in the desired state. The states of some resources (e.g., services or daemons) can also be checked explicitly to verify the task execution.

Use off-the-shelf testing libraries (6c). Each IaC language offer tools to be used for automated testing of IaC codes. For example, in Puppet, the test cases for classes and resource types should be developed using its testing framework (i.e., rspec). Tools like Vagrant allows for easy creation of virtual test environments compatible with production environments.

Monitors your environment (6d). Practitioners recommend continuous monitoring of the environment managed by IaC. This practice can be supported by APIs and query languages to retrieve the state of the environment automatically. For example, PuppetDB collects data generated by Puppet. The collected data can be queried using a REST API. For similar purposes, Puppet provides a module for collecting metrics about the managed environment.

5.3.7. Practice 7 - Document little but well

This category includes the guidelines for appropriately document IaC programs.

Code as documentation (7a). Source code should be treated as documentation. This practice ensures that documentation is always current because it is part of the code and stored in a central repository. The inclusion of additional

configuration instructions for users in a manual is discouraged as it can lead to infrastructure-documentation inconsistencies and non-reproducible environments.

Use document templates (7b). Templates and tools can be used to produce consistent documents. For example, as the roles are created to be shared across different projects typically using Ansible Galaxy, the recommendation is to document the roles consistently and sufficiently by using the template generated by Ansible Galaxy.

5.3.8. Practice 8 - Organize repositories well

This category of the best practices focuses on the proper organization of the IaC code repositories to foster improved and secured collaboration among users of repositories and to improve the understandability and predictability of the repository.

Modularize repositories (8a). The general recommendation is to use a single versioned-control IaC code repository (per organization) separated from the application source code repository. Chef considers cookbooks as standalone, self-contained applications, and thus a separate repository for each cookbook is desired. In Puppet, a control repository is a version-controlled repository that stores code, data, and modules or references to locations in other repositories. Different repositories, per each artifact, can allow separate access and development cycles of artifacts, such as complex Puppet modules, global data about the organization, and Puppet profiles.

Use standard folder structures (8b). Each IaC language recommends a specific directory layout while allowing some variations for their IaC projects. The recommended structure for the top-level directory of an Ansible project consists of inventory directories, variable (group and host) directories, custom module and plugin directories, master (top-level) playbook rile, role playbook files, and role directories. A Chef project includes three main sub-directories: *cookbooks*, *data_bags*, and *policyfiles* (i.e., groups of cookbooks and settings for specific

systems). The recommended structure for a Puppet module includes *data*, *files*, *functions*, *hiera.yaml*, *lib*, *manifests*, *metadata.json*, *plans*, and *tasks*.

5.3.9. Practice 9 - Separate configuration data from code

This category concerns the practices that aim to improve the management of configuration data.

Use configuration datasource (9a). When the number of the managed components exceeds a certain amount, the recommendation is to use a separate storage system to keep configuration data about those components (e.g., user names and server IPs). Ansible and Chef use file systems and version-controlled repositories (i.e., inventory files and data bags), and Puppet has *Hiera*, a key-value data store. The data configuration stores can also avoid hard-coding parameter defaults and private data in IaC programs.

Modularize configuration data (9b). The configuration data can also be modularized to improve their maintainability and usage. The environments such as test, development, and production exhibit differences in the number and configurations of resources, and thus environment-specific data are needed. In general, the same set of IaC code scripts deploys these environments. Building a reusable IaC code requires making the code configurable for many sites/environments. Using a separate data source (e.g., *Ansible inventory* files) for each environment minimizes the errors that can occur due to mixing configuration data from different environments. Moreover, the host data can be grouped based on the hosts' roles (e.g., web server and database server) to improve the modularity of an inventory file. In Puppet, *Hiera hierarchy* can represent the hierarchical organization of an environment.

Select data sources wisely (9c). IaC languages provide different options to store configuration data, and there exist guidelines for making appropriate choices considering different trade-offs. In Ansible, an inventory file defines the nodes in one or more target environments. The number and the properties of the nodes

can be static or can change at runtime, for example, in elastic environments such as public clouds. Thus, the best practices are related to the content organization for maintaining modularity and separation of concerns. Dynamic inventories can help to separate different environments in a dynamic and loosely-coupled way. In Chef, both data bags and environment files can store configuration data. Compared to data bags, accessing data from environment files incurs little or no overhead and thus is preferred over data bags for environment storing specific settings. While an environment file is a natural place to keep the server IP addresses, using service discovery tools is recommended due to the dynamic nature of server IP addresses. Recipes use conditionals on environment names or attributes to apply different resources in different environments. As the attributes can be overridden per-environment basis, they can implement environment-specific changes without the complexity added by conditionals.

Use configuration templates (9d). The configuration template pattern is the best practice for managing configuration files. It is recommended to externalize all template variables as input parameters (i.e., defining a public API) to write a self-contained, reusable template without directly referring to the attributes of either of the IaC codes using the template. The best practice is to create the parameterized reusable templates using variables and conditional logic to cater to variations in the site-specific configurations, such as using *Apache* for test and production environment.

5.3.10. Practice 10 - Write secure code

This category concerns secure data and software management.

Isolate secrets from code (10a). The key recommendation is to isolate secrets (sensitive information) such as passwords and private ssh keys from IaC code. Then, the isolated secrets should be injected into the deployment workflow as necessary during its execution.

Protect your data at rest (10b). The sensitive data should be encrypted when they are stored. Each IaC language supports a *Vault* for isolating and storing

secrets securely.

Use facts from trusted sources (10c). IaC programs may use the information or facts about the environments to make decisions such as classifying nodes and selecting suitable modules to manage nodes. Some IaC systems, for example, Puppet, provides the mechanisms to collect and access facts securely (i.e., *$trusted* fact array).

Use standard secure coding practices (10d). IaC developers also recommend secure coding practices such as secure logging and the principle of least privilege. In Ansible, to prevent displaying or logging decrypted data, the tasks' output should be selectively logged. The executions of some tasks need special privileges like root user or some other user (e.g., installing software). Enforcing fine-grained access control at the levels of a task or a block of tasks instead of globally at playbook or role levels is also recommended. The best practices for writing Puppet tasks include following the secure programming practices related to a given task and use the task parameter meta-data to declare if it holds sensitive data explicitly.

IaC Best Practices

We identified 10 primary categories of IaC best practices, sub-categorized in 33 lower-level categories. The practices cover each of the key constructs/abstractions of IaC languages. They reflect both implementation issues (e.g., naming convention, style, formatting, and indentation) and design issues (e.g., design modularity, reusability, and customizability of the code units of the different languages).

5.4. RQ3: IaC Bad Practices

In this section, we present the IaC bad practices reported in the selected grey literature. Table 4 shows the categories and sub-categories of the IaC bad practices. For each sub-category, the sources, the number of related codes

(frequency of codes) in sources, and the number of atomic practices are also shown.

Table 4: Bad practice categories and their sources (#), code frequencies (C), and number of atomic practices (P)

Practice	Independent			Ansible			Chef			Puppet		
	#	**P**	**C**	#	**P**	**C**	#	**P**	**C**	#	**P**	**C**
1 Violation of IaC principles												
1a Not letting IaC do its work	S28	4	4	S54 S65	2	3	-	-	-	S34	1	1
1b Violating Idempotence	-	-	-	S54 S65	2	4	-	-	-	S38	2	2
1c Using non-reproducible images and environments	S6 S28 S31	2	3	-	-	-	-	-	-	-	-	-
2 Not writing IaC programs for people												
2a Violation of naming and styling conventions	S28	1	1	S51 S65	2	2	-	-	-	S38	2	2
2b Favor complexity	-	-	-	S54	2	2	S63	2	2	-	-	-
2c Insufficient modularity	-	-	-	S47 S48 S50 S65	4	9	S43 S63	3	5	S34 S38	2	3
3 Improper project organization												
3a Improper repository usage	-	-	-	-	-	-	-	-	-	S30 S34 S36 S38	3	5
3b Improper version control	S18 S28	2	2	-	-	-	S40 S42 S63	3	5	S34 S38	2	2
4 Insecure configuration data and coding practices												
4a Hard coding information	S28	1	1	S57	1	1	-	-	-	S58	1	1
4b Not using built-in security tools and mechanisms correctly	-	-	-	-	-	-	S63	1	1	S38	2	2

5.4.1. Practice 1 - Violation of IaC principles

This category includes the bad development practices that violate the basic principles of IaC.

Not letting IaC do its work (1a). All manual procedures should be automated using the IaC source code. Configuration code (logic) and the data have different cycles and dynamics. Thus, keeping data along with code (in the same control repository) is not recommended. IaC languages provide high-level abstractions for managing the configuration settings of specific resources, such as an abstraction for configuring Apache servers. Thus, the overuse of low-level general abstractions such as file and package tools is considered an anti-pattern. As IaC, Ansible DSL source code should document itself, and the overuse of inline comments requires undue maintenance efforts.

Violating idempotence (1b). The violation of the idempotence property and the gradual creation of configuration drifts due to ad-hoc changes not managed by IaC can result in non-reproducible environments. In Ansible, imperative modules such as COMMAND and SHELL that execute ad-hoc operating system commands can break idempotence. Thus, the execution of the tasks using such modules should be guarded using conditionals that check the state of the infrastructure and its components. In Puppet, imperative statements (such as the resource type EXEC to run ad-hoc OS commands) can break one of the key philosophies of the language: the declarative configuration model [30].

Using non-reproducible images and environments (1c). Images for application components are generally created by extending a base or foundation image. If the base images are crafted manually without explicit specifications (e.g., Docker files), and such images get lost, updating and using those base images becomes difficult. The environments can also become non-reproducible due to manual or external updates outside automated IaC workflows.

5.4.2. Practice 2 - Not writing IaC programs for people

This category consists of the bad development practices that reduce readability, understandability, and testability of IaC scripts.

Violation of naming and styling conventions (2a). Improper naming and inconsistent formatting styles hinder the readability and maintainability of the IaC programs. For example, Ansible syntax is YAML-based, and hence the violation of YAML style guidelines and inconsistent use of such styles reduces the readability of the code. Similarly, in Puppet, roles and profiles should not be named, contradicting the abstractions they offer (e.g., having a role for managing technology and a profile for managing a server type).

Favor complexity (2b). For each IaC languages, the use of the complex conditions with many branches are discouraged as they are general untestable and reduce the readability of the code. The use of some language features that lead to complex unexpected behaviors, are discouraged. Ansible uses handlers to monitor and respond to the changes in the resources managed. A common usage of handlers is for restarting services (daemons) after an update to their configurations. If a handler causes a changed state (in the target infrastructure or node) in a handler chain, it notifies the next handler. The chain stops if a handler fails or does not change a resource, leading to unexpected behaviors that are difficult to debug and identify. Chef provides a publish-subscribe model of notifying and reacting to resource changes. However, the overuse of resource changes notifications creates a complex management workflow, which is extremely hard to follow and verify.

Insufficient modularity (2c). Having a single playbook or a few multifaceted playbooks hinder the separation of concerns, leading to code that is unnecessarily hard to test and maintain and slowing down the deployment process. In Ansible, this bad practice entails complete playbook executions for each infrastructure change. Using tasks without or along with roles in playbooks is discouraged as it makes the code less reusable and modular. Mixing-up abstractions reduce

the readability and testability. Defining variables in inventory files without logically organizing them (e.g., using groups that reflect the hosts) makes the inventories less modular. In Chef, too many attributes in a cookbook indicate that the cookbook has too many responsibilities and lacks proper abstractions and modularity. This violation is evident when writing recipes that result in many unrelated configuration functions (i.e., God Recipe anti-pattern), which undermines the recipe modularity. A similar issue occurs in Puppet, where multiple manifests should decompose the configuration logic into focused manifests.

5.4.3. Practice 3 - Improper project organization

This category concerns the wrong usage of repositories and version control systems.

Improper repository usage (3a). Using a single directory to keep all Puppet modules is discouraged. It leads to mixing up locally developed modules and upstream modules. The lifecycles of individual modules can be different, and hence, multiple systems may deploy different versions of modules. The separation of key Puppet artifacts such as component modules, roles, profiles, and Hiera data is recognized as a best practice. However, this practice can lead to multiple repositories and branches with different codebases and different module versions. In Puppet, the data in the profile modules specify profile defaults. Keeping these data in the same repository of the code is discouraged. It makes it difficult to delegate the development of profiles and their defaults to different teams (e.g., profile team and module team).

Improper Version Control (3b). Forking a community module/library without wrapping it or copying and pasting code to implement the same application in multiple environments is not recommended. The use of non-versionable codes such as Chef roles is also discouraged as changes have global effects.

5.4.4. Practice 4 - Do not write secure code

This category concerns violations of security practices for data and software management.

Hard coding information (4a). . Postponing the separation of secrets from the code and hard-coding sensitive information makes the code less reusable, less customizable, and more vulnerable.

Not using built-in security tools and mechanisms correctly (4b). Puppet offers several security functions, such as secure fact access and a module for auto signing certificates. Not relying on this built-in support is a bad practice. However, the users should evaluate the security vulnerabilities of the selected tools or mechanisms before deciding to use them. For example, Chef provides data bags as a way to manage secrets within Chef. However, using data bags is discouraged as a single decryption key is used for all secret information, and that key is distributed to every node.

IaC Bad Practices

We identified 4 primary categories of IaC bad practices, sub-refined in 10 categories. While most of these practices concern design and implementation issues related to key constructs/abstractions of IaC languages, they also reflect the violations of the essential principles of IaC: idempotence of configuration code, separation of configuration code from configuration data, and infrastructure/configuration management as software development.

6. Discussion, Highlights, and Observations

This section compares our work with existing academic literature and provides some observations on performing systematic grey literature reviews.

6.1. Comparison with IaC Practices Reported in the Existing Academic Literature

As discussed in Section 3, several studies [30, 31] on IaC smells have used best/bad practices as the sources of the smells. However, these studies considered

only a subset of the atomic practices: 32 practices for Puppet [30], and a similar amount for Chef [31]. We also observed that the reported best/bad practices are related only to a subset of the constructs/concepts of a given IaC language. For example, in Spinellis et al. [30], the practices concerning Puppet constructs roles, profiles, tasks, configuration files, and *Hiera* (configuration datastore) have not been used. We found 10 and 4 primary categories of IaC best and bad practices and considered each construct in three different IaC languages. Thus, our findings imply that new types of IaC smells are recognized, categorized, and detected.

Our survey with the industrial researchers and practitioners revealed seven IaC bad practices and four best practices [12]. Interestingly, all of those practices are also part of our catalog, albeit a few are only indirectly related. We found that little documentation is a best practice as IaC code should act as the documentation, which reduces the potential for occurring inconsistencies between code and documentation. However, that survey states that IaC code poor documentation is a bad practice.

Rahman et al. [33] identifies seven security smells for IaC by qualitatively analyzing IaC (Puppet) code scripts, which also reflect the insecure coding practices. Our catalog includes only two of those seven smells: admin by default and hard-coded secrets. Smells such as empty passwords, suspicious comments, and weak cryptography algorithms have not been reported in our selected industrial studies. However, we found new security best practices: (1) secure logging (not showing decrypted secrets in logs); (2) using facts (node properties) from a secure in-memory data store (a secure fact array); and (3) explicitly indicating if the value of a parameter contains sensitive information using parameter meta-data.

By qualitatively analyzing defect-related commits, Rahman et al. [34] compiled a taxonomy of eight IaC defects: (1) violation of idempotency property; (2) misconfigurations; (3) inconsistencies between documentation and IaC code; (4) erroneous conditional logic; (5) missing or incorrect dependencies; (6) security vulnerabilities; (7) incorrect use of *service/daemon* resource; and (8) syntax

defects. Interestingly, each of these bugs was indicated by at least one of the best/bad practices in our catalog. For example, the improper use of imperative commands/modules (e.g., SHELL and COMMAND modules in Ansible, and BASH resource in Chef) can result in non-idempotent IaC code, and thus is discouraged. Hence, the violation of best practices and the application of bad practices are potentially good early indicators of bugs.

6.2. Observations on Systematic Grey Literature Reviews

In this section, we observe the major difficulties and potentials of conducting systematic grey literature reviews.

Assessing quality of Grey Literature. White literature typically conforms to a pre-specified format, including abstract, keywords, introduction, methodology, results, evaluation, and page limitation. However, this statement does not hold for grey literature review, where there are different unique types of sources such as blogs, white papers, slides, and language guides. It is also worth highlighting that we observed that extracting data from slide decks is even more challenging because they usually lack details and the corresponding video or audio presentations. Please consider that grey literature cannot rely on the criteria used to assess white literature sources (i.e., ranking, h-index, and acceptance rate with the relevant research community). We found that the additional content of grey sources such as comments, likes, dislikes, and sharing are good indicators of their quality. However, most sources in our study did not have sufficient auxiliary content. Therefore, we applied a set of quality assessment criteria that focus on the reputation of the venues, the expertise of authors, and the clarity of the content of the article to address these challenges of selecting quality grey articles (see Section 4.3).

Lessons learned from applying Natural Language Processing techniques. Applying NLP techniques for the automated analysis of grey literature can complement the manual qualitative analysis to strengthen its importance and relevance. We initially attempted to apply topic modeling and topological data analysis.

In topic modeling, the main idea is to extract the latent topics from the data in an automated and unsupervised way. We compared the obtained topics with the codes and groups resulted from qualitative analysis. Then, we established a mapping (groups and topics) that allowed us to confirm the qualitative analysis findings and further refine them.

Topological Data Analysis (TDA) presents the shape of the unstructured data through a topological network. This technique provides a map of all the data set points: the closer the points, the closer their meanings. In our study, the points in the topology graph represent distributions of words. Simultaneously, the clusters are treated and interpreted separately to find their meaning and compare the topic modeling and manual qualitative analysis findings.

Most selected grey literature contained both text and code. In some cases, there was no clear separation between text and code using a special notation, preventing programs from differentiating them correctly. Moreover, many articles contain different content types, such as practices, features, examples, and concepts. Thus, it was problematic to separate the content belonging to different topics pragmatically accurately. These deficiencies in the pre-processing of grey literature resulted in low-quality data and less meaningful and broad topics. Thus, we decided to rely on manual qualitative analysis solely. We claim that more research is needed to derive methodologies and guidelines for NLP and data-driven techniques in systematic grey literature reviews.

Investigation approaches for handling fast growing materials. As grey literature resources are generally published more often than white literature resources, we observe that methodologies and tools for coping with fast-growing grey literature would be valuable. For example, a tool to (semi-)automatically update the literature review results or predict their relevance. Such tools may also have implications on how the review results are recorded. For example, a taxonomy could also be codified as an ontology [55], which can be shared, reused, and semi-automatically updated.

7. Threats to Validity

In this section, we outline the threats to validity that may apply to our study.

7.1. Threats to External Validity

External validity concerns the applicability of a set of results in a more general context [46]. Since our primary studies are obtained from many online sources, our results and observations may be only partially applicable to the broad area of practices and general disciplines of IaC, hence threatening external validity.

There is a risk of having missed relevant grey literature because concepts related to those included in our search strings are differently named in such studies. Some studies may refer to patterns, anti-patterns, or smells instead of best/bad practices. To mitigate this, we have explicitly included all relevant synonyms and similar words in our search strings. We have also exploited the features offered by search engines, which naturally support considering related terms for all those contained in a search string. Items found using the search terms have been assessed thoroughly based on various dimensions of quality [56].

Finally, while there are many IaC tools, we could study only three IaC tools due to practicality. To partially mitigate this threat, we selected the three most relevant tools that the practitioners currently use. We recognize that a comparison between the three selected tools and other home-grown solutions would lead to additional insights. We leave, however, such an analysis for future research work.

7.2. Threats to Construct and Internal Validity

Construct validity concern the generalizability of the constructs under study, while internal validity concerns the validity of the methods employed to study and analyze data (e.g., the types of bias involved) [46].

We organized at least four feedback sessions during our systematic analysis. We analyzed the discussion following-up from each feedback session, and we exploited this qualitative data to fine-tune both our research methods and the

applicability of our findings. We also prepared an online appendix containing all artifacts we produced during our analysis, including the full list of sources, codes, groups, and distilled best/bad practices (see Section 4). We are confident that this can help make our results and observations more explicit and applicable in practice.

Furthermore, we adopted various triangulation rounds, inter-rater reliability assessment, and quality control factors (recall Section 4). We applied inter-rater reliability assessment in at least two phases, both for the pilot study and the final study, with the full set of sources: primary sources selection and coding process. We added new studies and codes in the respective stages – although without performing a further inter-rater assessment. All in all, the risk of observer bias is always present when using this method.

7.3. Threats to Conclusions Validity

Threats to conclusions validity concern the degree to which the study conclusions are reasonably based on the available data [46].

To mitigate this threat, we again exploited theme coding and inter-rater reliability assessment to limit observer bias and interpretation bias, with the ultimate goal of performing a sound analysis of the data we retrieved. Additionally, the conclusions drawn in this article were independently drawn by the different authors. They were then double-checked against the selected industrial studies and/or related studies in one of our feedback rounds.

Overall, we know that our empirical investigation is limited to analyzing the practitioners' perception distilled from the grey literature. These findings, however, are in line with the ones stemming from focus groups with DevOps [12], and we are currently working to complement those studies with a large-scale mining software repository [57] (GitHub) investigation of how DevOps treat infrastructure code.

8. Conclusions

DevOps is a family of tactics that accelerate the deployment and delivery of large-scale applications. DevOps automation is driven by infrastructure code: the series of blueprints laying out the application infrastructure, its dependencies, and middleware across a DevOps pipeline.

This paper investigated infrastructure code language/tools and best/bad practices from a practitioner perspective by addressing grey literature in the field, stemming from 67 selected sources, and systematically applying qualitative analysis. We distilled a taxonomy consisting of 10 and 4 categories of best and bad IaC practices, respectively. We believe that this catalog, along with the categorization we provide, can be valuable for both practitioners and researchers. The former can benefit from comprehensive guidelines of "do's and don'ts" when developing IaC scripts. The latter can find foundations for further research e.g., on IaC patterns and anti-patterns/smells, which is part of our future work.

Our findings reveal critical insights concerning the top languages and the best practices adopted by practitioners to address (some of) those challenges. Overall, the most direct conclusion stemming from our evidence is that the field of software maintenance, evolution, and security of IaC is in its infancy and deserves further attention. On the one hand, several best practices exist, but they mostly concern the complexities inherent within IaC. On the other hand, many challenges exist, such as conflicting best-practices, lack of testability, security/secrets management issues, and monitoring.

Our future research agenda is based on the main findings of our grey literature review. We plan to provide automated mechanisms to recommend when and how to apply best practices in IaC code and pinpoint the bad practices affecting it. Furthermore, we also plan to replicate our investigation by considering more different IaC tools.

Acknowledgments

This research has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825480 (H2020 SODALITE) and No 825040 (H2020 RADON). Fabio acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PZ00P2_186090 (TED).

Appendix A. Selected Industrial Studies

S1 Sam Guckenheimer. What is Infrastructure as Code?. Microsoft, 2017. https://docs.microsoft.com/en-us/azure/devops/learn/what-is-infrastructure-as-code

S2 Mike Chan. 15 Infrastructure as Code tools you can use to automate your deployments. Thorntech, 2017. https://www.thorntech.com/2018/04/15-infrastructure-as-code-tools/

S3 Stephen Bigelow. What is infrastructure as code?. TechTarget, 2019. https://searchitoperations.techtarget.com/definition/Infrastructure-as-Code-IAC

S4 Jafari Sitakange. Infrastructure as Code: A Reason to Smile. ThoughtWorks, 2017. https://www.thoughtworks.com/insights/blog/infrastructure-code-reason-smile

S5 Artem Starostenko. Infrastructure as Code Tutorial. Hacker Noon, 2018. https://medium.com/hackernoon/infrastructure-as-code-tutorial-e0353b530527

S6 Dan Merron. Infrastructure as Code (IaC): An Introduction. BMC Software, 2018. https://www.bmc.com/blogs/infrastructure-as-code/

S7 Christopher Null. Infrastructure as code: The engine at the heart of DevOps. TechBeacon, 2017. https://techbeacon.com/enterprise-it/infrastructure-code-engine-heart-devops

S8 Eric. Infrastructure as Code: Everything You Need to Know. CloudBees, 2018. https://www.cloudbees.com/blog/infrastructure-as-code/

S9 Vikram Nallamala. What Is Infrastructure as Code?. DZone,2018. https://dzone.com/articles/what-is-infrastructure-as-code-2

S10 Steve Strutt. Infrastructure as Code: Chef, Ansible, Puppet, or Terraform?. IBM, 2018. https://www.ibm.com/cloud/blog/chef-ansible-puppet-terraform

S11 Steve Strutt. Infrastructure as Code Accelerates Application Deployment. IBM, 2018. https://www.ibm.com/cloud/blog/infrastructure-as-code

S12 Campbell. Infrastructure as Code: Seems Easy, But Best Left to Experts. Rackspace, 2019. https://www.rackspace.com/en-gb/blog/infrastructure-as-code-seems-easy-but-best-left-to-experts

S13 Krunal Sabnis. Infrastructure-as-code. Qualibrate, 2018. https://www.qualibrate.com/blog/infrastructure-as-code

S14 Vikram Nallamala. How To Leverage Infrastructure-As-Code With DevOps Best Practices. IbexLabs, 2018. https://www.ibexlabs.com/leverage-iac-with-devops-best-practices/

S15 Red Badger Team. What's Infrastructure as Code (IaC)?. Red Badger, 2018. https://blog.red-badger.com/2018/7/23/whats-infrastructure-as-code-iac

S16 Mike Chan. Infrastructure as Code: 5 Reasons Why You Should Implement IaC Now. Thorntech, 2018. https://www.thorntech.com/2018/01/infrastructureascodebenefits/

S17 David Schmitt and Igor Galić. Hitchhiker's guide to testing infrastructure as/and code — don't panic!. Puppet, 2016. https://puppet.com/blog/hitchhikers-guide-to-testing-infrastructure-as-and-code/

S18 Mike Brooks. Complexity in Infrastructure as Code. OVO Energy, 2018. https://tech.ovoenergy.com/complexity-in-infrastructure-as-code/

S19 Ricardo Matsui. Infrastructure as Code at Tile. Medium, 2018. https://medium.com/tile-engineering/infrastructure-as-code-at-tile-da0be83682a2

S20 Tom McLaughlin. Serverless DevOps: Infrastructure As Code With AWS Serverless. ServerlessOps, 2018. https://www.serverlessops.io/blog/serverless-ops-infrastructure-as-code-with-aws-serverless

S21 Roy Feintuch. New Security Challenges with Infrastructure-as-Code and Immutable Infrastructure. The New Stack, 2018. https://thenewstack.io/new-security-challenges-with-infrastructure-as-code-and-immutable-infrastructure/

S22 Mike Chan. Infrastructure as Code: 6 best practices to get the most out of IaC.Thorn Technologies, 2018. https://www.thorntech.com/2018/02/infrastructure-as-code-best-practices/

S23 Raj Bissessar. DevOps and Infrastructure as Code. Medium, 2019. https://medium.com/faun/devops-automation-and-iac-c007c3c0d172

S24 Oded Nahum. Infrastructure as Code: Driving Software Delivery Performance. LinkedIn, 2018. https://www.linkedin.com/pulse/infrastructure-code-driving-software-delivery-oded-nahum

S25 Tasktop. Software-Defined: DevOps Automation using Infrastructure as Code. Tasktop Technologies, 2017. https://www.tasktop.com/blog/software-defined-it-automation-using-infrastructure-as-code/

S26 Brett Johnson. Introduction to Infrastructure as Code. Network Computing, 2018. https://www.networkcomputing.com/networking/introduction-infrastructure-code

S27 Will May. Self-testing infrastructure-as-code. OpenCredo, 2018. https://opencredo.com/blogs/self-testing-infrastructure-as-code/

S28 Andrey Adamovich. Patterns for Infrastructure-as-Code. SlideShare, 2015. https://www.slideshare.net/aestasit/patterns-for-infrastructureascode

S29 Badri N. Srinivasan. Eleven Continuous Delivery Anti-Patterns. DZone, 2018. https://dzone.com/articles/eleven-continuous-delivery-anti-patterns

S30 Jonah Horowitz. Configuration Management is an Antipattern. Hacker Noon, 2018. https://hackernoon.com/configuration-management-is-an-antipattern-e677e34be64c\\#.pjuxtl98h

S31 Walter Pindhofer. Introduction Part 1: CI/CD Antipatterns. Qualysoft, 2018. https://www.qualysoft.com/en/blog/introduction-part-1-cicd-antipatterns

S32 Aiman Najjar. A Look at Aurrent "infrastructure as code" Trends. Pythian Group, 2018. https://blog.pythian.com/look-current-infrastructure-code-trends/

S33 Andrew Beresford. Puppet Anti-patterns. GitHub, 2018. https://github.com/beezly/puppet-antipatterns-presentation

S34 Tim Birkett. Puppet Anti-Patterns, 2016. https://www.pysysops.com/2016/11/10/1123-Puppet-Anti-Patterns.html

S35 David Danzilio. Puppet Design Patterns: The Factory Pattern, 2016. http://blog.danzil.io/2016/05/20/puppet-design-patterns-factory.html

S36 Gary Larizza. Building a Functional Puppet Workflow Part 1: Module Structure. 2014. http://garylarizza.com/blog/2014/02/17/puppet-workflow-part-1/

S37 David Danzilio. Puppet Design Patterns. SlideShare, 2016. https://github.com/beezly/puppet-antipatterns-presentation

S38 PuppetLab. CS Best Practices Repository. GitHub, 2018. https://github.com/puppetlabs/best-practices

S39 PuppetLab. Best Practices: Roles & Profiles. SlideShare, 2016. https://www.slideshare.net/PuppetLabs/puppetconf-2016-puppet-best-practices-roles-profiles-gary-larizza-puppet

S40 Nick Rycar. Chef 101: The Road to Best Practice. Chef, 2017. https://blog.chef.io/chef-101-the-road-to-best-practices

S41 Chef. Chef Style Guide. GitHub, 2017. https://github.com/pulseenergy/chef-style-guide

S42 Virendra. Chef workflows: patterns and anti-patterns. LinkedIn, 2015. https://www.linkedin.com/pulse/chef-workflows-patterns-anti-patterns-virendra-bhalothia

S43 Eric Krupnik. Chef Cookbook Design Patterns. SlideShare, 2015. https://www.slideshare.net/EricKrupnik/chef-cookbook-design-patterns

S44 Stanisław Tuszyński. Chef best practices. Ragnarson, 2017. https://blog.ragnarson.com/2015/06/01/chef-best-practices.html

S45 Bruce Cutler. Testing your Chef Code: It's All About Confidence. Medium, 2017. https://medium.com/slalom-technology/testing-your-chef-code-its-all-about-confidence-fd4b9d969a7e

S46 Ansible Community. Tips and tricks. Ansible, 2019. https://docs.ansible.com/ansible/2.8/user_guide/playbooks_best_practices.html

S47 Michel Blanc. Laying out roles, inventories and playbooks. GitHub, 2017. https://leucos.github.io/ansible-files-layout

S48 Timothy Appnel. Inside Playbook Ansible Best Practices: The Essentials, Ansible, 2016. https://www.ansible.com/blog/ansible-best-practices-essentials

S49 Kat Dober. New to Ansible? Check Out Our Best Practices Guide. New Relic Software, 2018. https://blog.newrelic.com/engineering/ansible-best-practices-guide/

S50 Jiri Tyr. Best practices for ansible roles development. SlideShare, 2017. https://www.slideshare.net/jtyr/best-practices-for-ansible-roles-development

S51 Sebastian Gumprich. Ansible Best practices. T-Systems, 2018. https://blog.t-systems-mms.com/tech-insights/ansible-best-practices

S52 Manosh Malai. Ansible Best Practices. Wordpress, 2018. https://mydbops.wordpress.com/2018/04/07/ansible-best-practices/

S53 Engin Yöyen. Ansible Best Practices. GitHub, 2017. https://github.com/enginyoyen/ansible-best-practises

S54 Julio Villarreal Pelegrinoa. Ansible and Ansible Tower: best practices from the fields. 2018. https://juliosblog.com/ansible-and-ansible-tower-best-practices-from-the-field/

S55 Soroush Atarod. Ansible Best Practices explained. 2018. https://www.infinitypp.com/ansible/best-practices/

S56 Will Robinson. The Anatomy of an Ansible Playbook. 2017. https://oznetnerd.com/2017/04/09/anatomy-ansible-playbook/

S57 Andreas Sommer. Ansible Best Practices. Ansible best practices, 2017. https://andidog.de/blog/2017-04-24-ansible-best-practices

S58 Gary Larizza. Hiera, data and Puppet code: your path to the right data decisions. Puppet, 2017. https://puppet.com/blog/hiera-data-and-puppet-code-your-path-right-data-decisions/

S59 Lori Macvittie. The Dark Side of Infrastructure as Code. DevOps, 2014. https://devops.com/dark-side-infrastructure-code/

S60 Andrés Riancho. Secure infrastructure as code:How I built w3af.org. OWASP, 2015. https://owasp.org/www-pdf-archive/8-Secure_infrastructure_as_code_-_How_I_built_w3af.org_-_v0.4.pdf

S61 Sudhi Seshachala. Ansible Best Practices. DevOps, 2015. https://devops.com/puppet-best-practices/

S62 David Marquis. A few Puppet best practices. Radiant3 Productions, 2013. http://blog.radiant3.ca/2013/09/17/a-few-puppet-best-practices/

S63 Chef Community. Emerging Anti Patterns in Chef. GitHub, 2015. https://github.com/chef-boneyard/community-summits/wiki/Seattle2015-Emerging-Anti-Patterns

S64 Glenn Poston. Puppet Best Practices: Environment specific configs, 2013. http://www.glennposton.com/posts/puppet_best_practices__environment_specific_configs

S65 Oteemo Team. Organizing Ansible. Oteemo, 2019. https://oteemo.com/organizing-ansible/

S66 Chef Team. Patterns To Follow. Chef.io, 2019. https://docs.chef.io/ruby/#patterns-to-follow/

S67 Puppet Team. The Puppet language style guide. Puppet, 2019. https://puppet.com/docs/puppet/7.4/style_guide.html

References

[1] L. J. Bass, I. M. Weber, L. Zhu, DevOps - A Software Architect's Perspective., SEI series in software engineering, Addison-Wesley, 2015.

[2] M. Artac, T. Borovsak, E. D. Nitto, M. Guerriero, D. A. Tamburri, Model-driven continuous deployment for quality devops., in: D. Ardagna, G. Casale, A. van Hoorn, F. Willnecker (Eds.), QUDOS@ISSTA, ACM, 2016, pp. 40–41.

[3] K. Morris, Infrastructure As Code: Managing Servers in the Cloud, Oreilly & Associates Incorporated, 2016.

[4] 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.

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

[6] M. Jarschel, Network function virtualization: Towards the commoditization of middle boxes (2013-11-05).

[7] 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.

[8] P. Lipton, D. Palma, M. Rutkowski, D. A. Tamburri, Tosca solves big problems in the cloud and beyond!, IEEE cloud computing 5 (2) (2018) 37–47.

[9] L. Hochstein, R. Moser, Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way, " O'Reilly Media, Inc.", 2017.

[10] J. Loope, Managing infrastructure with puppet: configuration management at scale, " O'Reilly Media, Inc.", 2011.

[11] M. Marschall, Chef infrastructure automation cookbook, Packt Publishing Ltd, 2015.

[12] 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.

[13] A. Rahman, R. Mahdavi-Hezaveh, L. Williams, A systematic mapping study of infrastructure as code research, Information and Software Technology 108 (2019) 65–77.

[14] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani, Tosca lightning: An integrated toolchain for transforming tosca light into production-ready deployment technologies, in: N. Herbaut, M. La Rosa (Eds.), Advanced Information Systems Engineering, Springer International Publishing, 2020, pp. 138–146.

[15] S. Dalla Palma, D. Di Nucci, F. Palomba, D. A. Tamburri, Toward a catalog of software quality metrics for infrastructure code, Journal of Systems and Software 170 (2020) 110726.

[16] S. Dalla Palma, D. Di Nucci, D. A. Tamburri, Ansiblemetrics: A python library for measuring infrastructure-as-code blueprints in ansible, SoftwareX 12 (2020) 100633.

[17] S. Dalla Palma, D. Di Nucci, F. Palomba, D. A. Tamburri, Within-project defect prediction of infrastructure-as-code using product and process metrics, IEEE Transactions on Software Engineering (2021) 1–1.

[18] I. Kumara, Z. Vasileiou, G. Meditskos, D. A. Tamburri, W.-J. Van Den Heuvel, A. Karakostas, S. Vrochidis, I. Kompatsiaris, Towards semantic detection of smells in cloud infrastructure code, in: Proceedings of the 10th International Conference on Web Intelligence, Mining and Semantics, WIMS 2020, Association for Computing Machinery, 2020, pp. 63–67.

[19] N. Borovits, I. Kumara, P. Krishnan, S. D. Palma, D. Di Nucci, F. Palomba, D. A. Tamburri, W.-J. van den Heuvel, Deepiac: Deep learning-based linguistic anti-pattern detection in iac, in: Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation, MaLTeSQuE 2020, Association for Computing Machinery, 2020, pp. 7–12.

[20] G. Schermann, S. Zumberi, J. Cito, Structured information on state and evolution of dockerfiles on github, in: Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18, ACM, 2018, pp. 26–29.

[21] T. Dai, A. Karve, G. Koper, S. Zeng, Automatically detecting risky scripts in infrastructure code, in: Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, Association for Computing Machinery, 2020, pp. 358–371.

[22] T. Sotiropoulos, D. Mitropoulos, D. Spinellis, Practical fault detection in puppet programs, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, Association for Computing Machinery, 2020, pp. 26–37.

[23] J. Sandobalín, E. Insfran, S. Abrahão, On the effectiveness of tools to support infrastructure as code: Model-driven versus code-centric, IEEE Access 8 (2020) 17734–17761.

[24] M. S. Islam Shamim, F. Ahamed Bhuiyan, A. Rahman, Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices, in: 2020 IEEE Secure Development (SecDev), 2020, pp. 58–64.

[25] M. M. Hasan, F. A. Bhuiyan, A. Rahman, Testing practices for infrastructure as code, in: Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing, LANGETI 2020, Association for Computing Machinery, 2020, pp. 7–12.

[26] A. Rahman, E. Farhana, L. Williams, The 'as code' activities: development anti-patterns for infrastructure as code, Empirical Software Engineering 25 (5) (2020-09-01) 3430–3467.

[27] A. Rahman, M. R. Rahman, C. Parnin, L. Williams, Security smells in ansible and chef scripts: A replication study, ACM Transactions on Software Engineering and Methodology (TOSEM) 30 (1) (2021-01).

[28] R. Opdebeeck, A. Zerouali, C. Velázquez-Rodríguez, C. D. Roover, Does infrastructure as code adhere to semantic versioning? an analysis of ansible role evolution, in: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2020, pp. 238–248.

[29] S. Kokuryo, M. Kondo, O. Mizuno, An empirical study of utilization of imperative modules in ansible, in: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), 2020, pp. 442–449.

[30] T. Sharma, M. Fragkoulis, D. Spinellis, Does your configuration code smell?, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, ACM, 2016, pp. 189–200.

[31] 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), 2018, pp. 220–228.

[32] 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.

[33] A. Rahman, C. Parnin, L. Williams, The seven sins: security smells in infrastructure as code scripts, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 164–175.

[34] A. Rahman, E. Farhana, C. Parnin, L. Williams, Gang of eight: A defect taxonomy for infrastructure as code scripts, in: Proceedings of the 42nd

International Conference on Software Engineering, ICSE, Vol. 20, 2020, pp. 752–764.

[35] A. Rahman, A. Partho, P. Morrison, L. Williams, What questions do programmers ask about configuration as code?, in: Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering, RCoSE '18, ACM, 2018, pp. 16–22.

[36] L. Leite, C. Rocha, F. Kon, D. Milojicic, P. Meirelles, A survey of devops concepts and challenges, ACM Computing Surveys (CSUR) 52 (6) (2019-11).

[37] D. WeerasiriTaxonomyCloud, M. C. Barukh, B. Benatallah, Q. Z. Sheng, R. Ranjan, A taxonomy and survey of cloud resource orchestration techniques, ACM Computing Surveys (CSUR) 50 (2) (2017-05).

[38] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann, A systematic review of cloud modeling languages, ACM Computing Surveys (CSUR) 51 (1) (2018-02).

[39] S. Keele, et al., Guidelines for performing systematic literature reviews in software engineering, Tech. rep., Technical report, Ver. 2.3 EBSE Technical Report. EBSE (2007).

[40] V. Garousi, M. Felderer, M. V. Mäntylä, Guidelines for including grey literature and conducting multivocal literature reviews in software engineering, Information and Software Technology 106 (2019) 101–121.

[41] J. Soldani, D. A. Tamburri, W.-J. Van Den Heuvel, The pains and gains of microservices: A systematic grey literature review, Journal of Systems and Software 146 (2018) 215–232.

[42] R. Verdecchia, I. Malavolta, P. Lago, Guidelines for architecting android apps: A mixed-method empirical study, in: 2019 IEEE International Conference on Software Architecture (ICSA), 2019, pp. 141–150.

[43] V. Garousi, B. Küçük, Smells in software test code: A survey of knowledge in industry and academia, Journal of systems and software 138 (2018) 52–81.

[44] C. Islam, M. A. Babar, S. Nepal, A multi-vocal review of security orchestration, ACM Computing Surveys (CSUR) 52 (2) (2019-04).

[45] B.-J. Butijn, D. A. Tamburri, W.-J. v. d. Heuvel, Blockchains: a systematic multivocal literature review, ACM Computing Surveys (CSUR) 53 (3) (2020) 1–37.

[46] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[47] J. Saldaña, The coding manual for qualitative researchers, Sage, 2015.

[48] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. Chue Hong, M. Davis, R. T. Guy, S. H. D. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, P. Wilson, Best practices for scientific computing, PLoS Biol 12 (1) (2014-01) 1–7.

[49] G. Wilson, J. Bryan, K. Cranston, J. Kitzes, L. Nederbragt, T. K. Teal, Good enough practices in scientific computing, PLoS computational biology 13 (6) (2017-06) 1–20.

[50] M. Taschuk, G. Wilson, Ten simple rules for making research software more robust, PLoS computational biology 13 (4) (2017-04) 1–10.

[51] M. Graff, K. R. Van Wyk, Secure coding: principles and practices, " O'Reilly Media, Inc.", 2003.

[52] J. Varia, Best Practices in Architecting Cloud Applications in the AWS Cloud, John Wiley & Sons, Ltd, 2011, Ch. 18, pp. 457–490. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470940105.ch18.

[53] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, M. Di Penta, An empirical characterization of bad practices in continuous integration, Empirical Software Engineering 25 (2) (2020-03-01) 1095–1135.

[54] I. Abal, J. Melo, Ş. Stănciulescu, C. Brabrand, M. Ribeiro, A. Wąsowski, Variability bugs in highly configurable systems: a qualitative analysis, ACM Transactions on Software Engineering and Methodology (TOSEM) 26 (3) (2018) 1–34.

[55] N. Guarino, D. Oberle, S. Staab, What is an ontology?, in: Handbook on ontologies, Springer, 2009, pp. 1–17.

[56] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: Proceedings of the 18th international conference on evaluation and assessment in software engineering, Citeseer, 2014, p. 38.

[57] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian, The promises and perils of mining github, in: Proceedings of the 11th working conference on mining software repositories, ACM, 2014, pp. 92–101.