Static Analysis of Infrastructure as Code: a Survey

Michele Chiari
DEIB, Politecnico di Milano
Milano, Italy
michele.chiari@polimi.it

Michele De Pascalis
DEIB, Politecnico di Milano
Milano, Italy
michele.depascalis@mail.polimi.it

Matteo Pradella
DEIB, Politecnico di Milano
IEIIT, Consiglio Nazionale delle Ricerche
Milano, Italy
matteo.pradella@polimi.it

***Abstract*—The increasing use of Infrastructure as Code (IaC) in DevOps leads to benefits in speed and reliability of deployment operation, but extends to infrastructure challenges typical of software systems. IaC scripts can contain defects that result in security and reliability issues in the deployed infrastructure: techniques for detecting and preventing them are needed. We analyze and survey the current state of research in this respect by conducting a literature review on static analysis techniques for IaC. We describe analysis techniques, defect categories and platforms targeted by tools in the literature.**

***Index Terms*—infrastructure as code, cloud computing, static analysis, model checking, verification, survey**

I. Introduction

Infrastructure as Code (IaC) allows for defining, deploying, managing, and orchestrating computing systems in a fully automated way, based on a textual description of their supporting infrastructure and configuration [1]. IaC is part of a larger trend toward the automation of both development and system implementation operations known as *DevOps* [2]. The integration with widely-used cloud providers offered by IaC platforms makes IaC particularly beneficial for systems leveraging *Cloud Computing*. Moreover, the benefits of using IaC instead of manual deployments are reduced deployment times, better repeatability, and easier system maintenance, update and management.

On the one hand, representing infrastructure and configuration as code avoids possible errors introduced by human maintainers during deployment operations. On the other hand, IaC presents features similar to software code [3]: IaC codebases may reach considerable sizes, and are subject to modifications by different maintainers. Thus, IaC scripts may present similar pitfalls, such as bugs and defects that cause deployment failures or the deployment of defective configurations, that may cause availability, security, performance, or reliability problems.

To prevent and correct such issues, software engineering techniques similar to those developed for software code are increasingly applied to IaC. These techniques can be divided in dynamic approaches, such as testing and monitoring, which leverage actual deployment executions, and static approaches, which only rely on the analysis of syntactic or semantic features of IaC. Static analysis can be particularly beneficial for IaC, because it does not require deployment of possibly defective infrastructure, that can be expensive and time-consuming, due to the need of implementing isolated canary environments.

In this paper, we give an overview of recent efforts to develop and apply static analysis techniques to IaC. We conduct a literature review guided by the following research questions:

- RQ1 What are the tools available for static verification and validation of IaC?
- RQ2 What are the techniques used by existing tools?
- RQ3 What kinds of properties are checked by existing tools?
- RQ4 What IaC languages are targeted by existing tools?

These questions are aimed at gathering an overview of the current state of the art on static techniques for the analysis and detection of anomalies in IaC, to inform researchers on the current trends and highlight research gaps that may lead to future research directions.

As a result, we identify two large categories for IaC static analysis techniques:

- Those that operate on mostly syntactic features, and rely on code-smell detection or machine learning and data mining.
- Those that analyze possible behaviors of the infrastructure in different deployment phases, which rely on automated verification techniques such as model checking.

We describe the most important works in each category and compare them. Since a considerable number of papers deals with code smells, we summarize them systematically, to give an overview of defects that may affect IaC.

Related Work. We observe a lack of reviews covering IaC verification literature. A. Rahman et al. [4] do a systematic mapping study of IaC research. They analyze research on empirical analysis and testing, without focusing on static analysis. A. Alnafessah et al. [5] survey work on DevOps quality assurance. They treat verification in DevOps, but mostly referencing software analysis techniques that are not specific to IaC.

Paper Structure. In Sect. II we briefly introduce IaC; in Sect. III we describe our review methodology; in Sect. IV and

This project has received funding from the European Union's Horizon 2020 programme under grant agreements No 101000162 (PIACERE) and 825480 (SODALITE).

V we review papers by classifying them in two categories; in Sect. VI we answer the research questions and conclude the paper.

II. Background: Infrastructure as Code

IaC is the production of machine-readable textual descriptions of—often cloud—infrastructure that can be automatically executed to deploy and manage computing systems. IaC platforms provide domain-specific languages for IaC scripts, and offer plugins for integration in cloud service providers and virtualization tools. Different IaC tools cover different phases of deployment.

Tools such as Ansible[1], Chef[2], and Puppet[3] manage the software configuration of computing nodes. They can deploy software and containers, configure and manage them. Thus, they often support embedding of system configuration languages such as shell scripts. Ansible and Chef scripts explicitly describe actions required to reach the target configuration, and can be considered *imperative*, while Puppet describes it in a *declarative* way. Actions performed by Puppet are *idempotent*, i.e., executing them repeatedly any number of times yields the same results as executing them once.

Platforms such as AWS CloudFormation[4] and Terraform[5] focus on provisioning. They read declarative specifications describing virtual machines and networks connecting them, and synthesize a plan for creating, updating or deleting them. The plan is then executed to provision the infrastructure. Deployment tools like Cloudify[6] can be used to deploy applications on already-provisioned environments.

While each tool targets a different IaC language, the TOSCA [6] language has been introduced to standardize them. TOSCA allows for a declarative specification of the infrastructure topology as *services templates*, describing concrete or virtual computing nodes, the services they run, and networks connecting them. Node types can be defined to describe classes of nodes offering certain capabilities, which can be referenced by other nodes. Deployment plans for TOSCA service templates are created and executed by deployment tools (e.g., Cloudify) and orchestrators (e.g, xOpera[7]).

For a more complete introduction to IaC, we refer to [1].

III. Methodology

We performed our literature review by searching the main search engines for scientific literature.

We defined the search strings by gathering keywords from the research questions, and by considering possible synonyms and related terms. This resulted in a query of the form $A \wedge B$ where $A =$ Infrastructure as Code $\vee$ IaC $\vee$ TOSCA $\vee$ Terraform $\vee$ Cloudify $\vee$ Puppet $\vee$ Chef $\vee$ Ansible $\vee$ Cloud, and $B =$ Verification $\vee$ Validation $\vee$ Validity $\vee$ Static Analysis $\vee$ Model Checking $\vee$ SMT $\vee$ Code Smell $\vee$ Linter.

We used the following search engines: ACM Digital Library, Elsevier Scopus, Clarivate Web of Science, IEEE Xplore Digital Library, and Google Scholar.

After gathering search results, we selected relevant entries by including only those describing IaC static-analysis techniques, and by excluding:

- papers on verification of cloud infrastructure not specified through IaC;
- papers employing dynamic techniques such as testing and monitoring;
- position papers with limited technical content.

We sorted the results from search engines by relevance to the query, and stopped after finding 20 consecutive results that would be excluded according to these criteria.

The search on Google Scholar was performed by the second author, and searches on other engines by the first one. Inclusion of papers selected during these searches was subsequently rediscussed by both authors.

IV. Code Smells and Defect Predictors

The need to apply software engineering techniques typical of mainstream programming languages to IaC was first highlighted in Y. Jiang and B. Adams [3]. The authors conducted an empirical study on IaC code bases targeting Chef and Puppet, gathered from the OpenStack project repositories. They found out that IaC code takes up a median proportion of 11% of code in a repository, which is substantially larger than build files. Moreover, the distribution of monthly changes to IaC files is similar to that of production code files, although IaC files show a smaller *churn*.[8] Thus, IaC deserves the same attention as ordinary software in studying, finding and preventing bugs.

In this section, we survey tools and techniques that find potential bugs in IaC scripts by relying mostly on the analysis of syntactic features thereof. This category includes linters, detectors of code smells and metrics that correlate with defects in IaC scripts.

A. Code Smells

Code smells are code features and patterns that, while not being erroneous *per se*, indicate bad code quality, and strongly correlate with coding mistakes and bugs. The concept was originally introduced by K. Beck and M. Fowler [7], and has been extensively studied by empirical software engineering researchers, who thoroughly categorize code smells for various programming languages. We summarize IaC smells found in the literature in Table I.

In the following, defect detectors are evaluated in terms of *precision*, i.e., the proportion of true positives over all detected issues; and *recall*, i.e., the proportion of detected issues over all actual issues. The harmonic mean of precision and recall is called *F-score* [8].

[1] https://www.ansible.com/
[2] https://www.chef.io/
[3] https://puppet.com/
[4] https://aws.amazon.com/cloudformation/
[5] https://www.terraform.io/
[6] https://cloudify.co/
[7] https://github.com/xlab-si/xopera-opera

[8] *Churn* is a measure of the size of code changes in a software project. Y. Jiang and B. Adams [3] measure it in lines of code changed in a commit.

The first empirical study on code smells in IaC was carried out by T. Sharma et al. [9] on 4,621 GitHub repositories containing Puppet scripts. The authors manually identified 24 code smells, divided in *implementation smells* (I in Table I), which affect coding style and formatting issues, and *design smells* (D), which affect the use of abstraction mechanisms offered by Puppet scripts. They used the *Puppet-Lint* tool augmented with custom rules to detect implementation smells, and developed a tool called *Puppeteer*[9] to detect design smells, mostly based on code metrics specifically developed for each smell. They found out that the most common implementation smells are "improper alignment", "improper quote usage", and "long statement", while the most common design smells are "insufficient modularization" and "multifaceted abstraction". This study, however, does not investigate the degree of correlation between code smells and actual bugs in IaC scripts.

The results by T. Sharma et al. [9] are extended to Chef scripts by J. Schwarz et al. [10]. They classify code smells in *technology agnostic* (A in Table I), which are applicable to any IaC platform without altering the detection method substantially, *technology dependent* (E), which can be applied to different platforms after modifying the detection method, and *technology specific* (S), which can be applied to a single IaC language. They classified code smells from [9] into these categories and provided more general definitions for technology-agnostic ones. Moreover, they introduced five new technology-agnostic or dependent smells and two Chef-specific ones. The authors then augmented the Chef linting tool Foodcritic[10] to detect the smells they studied, and used it to analyze Chef scripts from the repository of an industrial partner (35 cookbooks) and the official Chef repository (3200 cookbooks). They found out that the most frequently occurring smells are "improper alignment", "long statement" and "misplaced attribute", which agrees with the findings by T. Sharma et al. [9] in Puppet scripts. Conversely, the "improper quote usage" smell is not as frequent in Chef as in Puppet.

A. Rahman et al. [11] conduct a study on code smells focused on security issues in Puppet scripts. They identify 7 security smells by applying *descriptive coding* [12], a qualitative analysis technique, to 1,726 Puppet scripts gathered from projects by Mozilla, OpenStack, and Wikimedia. Each identified smell is associated to a security weakness from the Common Weakness Enumeration (CWE) [13]. Such smells include issues common in contexts other than IaC: "admin by default" increases the attack surface by granting excessive access rights to a system, "hard-coded secret" exposes user names and passwords that should not be revealed, and "use of HTTP without TLS" and "use of weak cryptography algorithms" undermine secrecy and integrity of communications by using outdated technologies. The authors develop SLIC[11], a tool that detects security smells in Puppet scripts by means of rules based on string patterns. They evaluate SLIC on a dataset of 140 Puppet scripts with manually-identified code smells. SLIC's overall precision and recall turn out to be very high (both 0.99). The authors then conduct an empirical study by running SLIC on 15,232 Puppet scripts gathered from GitHub, Mozilla, OpenStack, and Wikimedia repositories. The most frequent smell is "hard-coded secret", present in respectively 21.9%, 9.9%, 24.8%, and 17.0% of the above mentioned sources. Other frequent smells are "suspicious comment" and "use of HTTP without TLS". To evaluate the actual impact of security smells, they surveyed practitioners by submitting 1,000 bug reports on the detected occurrences, obtaining a 21.2% response rate. Practitioners agreed on 69.8% of the bugs, with a peak agreement of 84.6% for "use of weak cryptography algorithms" and more than 75% for "use of HTTP without TLS". The smells with least agreement are "suspicious comment", slightly above 25%, and "default admin" and "empty password", both slightly above 50%.

A further empirical study is done by F. A. Bhuiyan and A. Rahman [14] to investigate correlation between the security smells identified in [11]. The work is carried out on the same dataset and using the same tool as [11]. The smell pairs that are most frequently co-located are "hard-coded secret" and "suspicious comment" (which co-locate in up to 11% of the scripts in one dataset), and "hard-coded secret" and "HTTP without TLS" (up to 16%). The authors also investigate which metrics correlate with security smells: metrics that correlate most strongly are the *number of lines of code* and *configuration attributes*, and the presence of *hard-coded strings*.

A replication study is done in A. Rahman et al. [15], where the security smells of [11] are extended to Ansible and Chef scripts. They identify two more smells, namely "no integrity check" and "missing default case statement", and develop SLAC, a tool for identifying these smells plus those from [11] in Ansible and Chef scripts. SLAC is based on pattern-matching rules, like SLIC, and is evaluated in two rounds on two datasets of Ansible and Chef scripts, where smell instances have been manually identified through *closed coding* [12]. The evaluation shows that SLAC has very high precision and recall, both between 0.9 and 1.0 on Ansible and Chef scripts. An empirical study is also conducted on 4,253 Ansible and 6,070 Chef scripts gathered from OpenStack and GitHub repositories. The results are similar to those for Puppet scripts in [11]: the most frequent smell is "hard-coded secret", which affects from 6.8% of Chef scripts from GitHub to 22.4% Ansible scripts from OpenStack, followed by "suspicious comment" and "use of HTTP without TLS". A practitioner survey is also conducted by sending bug reports for 500 Ansible and 500 Chef smell occurrences. The findings are again similar to those for Puppet scripts: "use of weak cryptography algorithms" has an agreement well above 75%, and "use of HTTP without TLS" also has a high agreement (100% for Ansible and a little less than 75% on Chef). "Suspicious comment" still has a very low agreement for Chef scripts (less than 30%), but it is higher for Ansible (above 75%). Additionally, "unrestricted IP address" and "missing default case statement" have a 100% agreement on Chef scripts. We must note, however, that the

[9]https://github.com/tushartushar/Puppeteer

[10]Foodcritic custom rules available at https://github.com/swc-rwth/InfrastructureAsCodeSmells. We represent this as Foodcritic† in Table I.

[11]https://github.com/akondrahman/IacSec

response rate from practitioners was low (9.4%), which might undermine the significance of these results.

I. Kumara et al. [16] develop a tool[12] for detecting code smells in TOSCA service templates, as part of the SODALITE project[13]. They devise a semantic model of a TOSCA deployment, and create SPARQL [17] rules based on it to detect smells. The smells they detect are mostly security-related. The tool is evaluated on a case study.

A. Rahman et al. [18] apply descriptive coding to 1,448 defect-related commits from 61 OpenStack repositories containing Puppet scripts and generate a taxonomy of defects. They identify eight categories:

- *Conditional*: defects that originate from wrong logic in conditional choices;
- *Configuration Data*: erroneous—possibly hard-coded—configuration settings;
- *Dependency*: external artifacts required for the script to work are missing;
- *Documentation*: code comments and documentation contain outdated or wrong information;
- *Idempotency*: a script is *idempotent* if executing it repeatedly any number of times yields the same results as executing it one time. If this is not the case, the script violates this property.
- *Security*: confidentiality, integrity or availability are compromised for the provisioned system;
- *Service*: defects related to improper provisioning of computing services;
- *Syntax*: the syntax of the IaC language is violated.

Although these are not—strictly speaking—code smells, we include this work in this section because ACID, the tool they develop to detect and categorize defects, is based on similar techniques. ACID analyzes commit messages and diffs by using a set of pattern-matching rules manually devised by the authors. An evaluation of ACID on an oracle dataset yields average precision and recall resp. of 0.84 and 0.96. An empirical study on commits from 291 repositories from GitHub, Mozilla, OpenStack and Wikimedia shows that "configuration data" is the most frequent defect category, followed by "syntax" and "dependency".

B. Methods based on Data-Mining

While IaC smells are devised manually by authors of works described in Sect. IV-A, in this section we analyze works where script features indicating potential defects are obtained through systematic qualitative analyses or data and code mining techniques, which are then utilized to create detection tools.

A. Rahman and L. Williams [20] collect 2,259 Puppet scripts from repositories by Mozilla, OpenStack and Wikimedia Commons, and apply qualitative analysis to their commits to identify the ones related to IaC script defects. Then, features of defective scripts are mined through the *Bag of Words* (BOW) [21] and *Term Frequency-Inverse Document Frequency* (TF-IDF) [22] techniques, and the most relevant ones are selected through *Principal Component Analysis* (PCA) [8]. Categories are derived from such text features through a qualitative analysis by *Strauss-Corbin Grounded Theory* (SGT) [23]. They identify three categories: *filesystem operations*, *infrastructure provisioning*, and *managing user accounts*. Predictors for these categories are built by training *Random Forest* [8] statistical learners, which are evaluated by applying *10-fold cross-validation* [8] on the same dataset. The resulting median F-scores range from 0.70 to 0.74.

More fine-grained categories are obtained by A. Rahman and L. Williams in [24], where they apply constructivist grounded theory to defect-related commit messages and diffs to identify source-code properties of defective scripts. The dataset consists of 2,439 Puppet scripts from Mirantis, Mozilla, OpenStack and Wikimedia repositories. Some of the properties they find have also been identified as code smells (cf. Sect. IV-A). The properties that correlate the most with defects are the number of lines of code and hard-coded strings. The authors then build predictors using five different statistical learners and evaluate them using 10-fold cross-validation, obtaining F-scores between 0.67 and 0.70 for the best techniques. Predictors are also compared with implementation smells by T. Sharma et al. [9], finding out that predictors are better in precision but worse in recall.

N. Borovits et al. [25] developed DeepIaC, a tool that uses deep learning to detect anti-patterns in Ansible scripts. DeepIaC classifies scripts based on whether they contain anti-patterns by means of Convolutional Neural Networks [26] trained on a dataset of scripts with artificially-inserted bugs. An empirical evaluation on 18,286 scripts taken from 38 GitHub repositories shows that DeepIaC detects such artificial bugs with an accuracy ranging from 0.79 to 0.92.

Code and process metrics are evaluated by S. Dalla Palma et al. [27] as features for machine-learning-based detection of defects in IaC scripts. The authors mine GitHub repositories and gather a dataset of 104 repositories containing defective Ansible scripts [28]. Then, they automatically extract from them 108 features taken from previous work on process metrics for general software [29], [30] and code metrics specific to IaC [24]. These features are used to train predictors consisting of a feature selection, a data balancing, a data normalization, and a classification phase, using different combinations of techniques for each phase. They find out that Random Forest is by far the best predictor, scoring first in terms of accuracy and recall on 98 repositories out of 104. Moreover, IaC-specific metrics greatly outperform other metrics as prediction features. The resulting predictors are used in the defect-prediction framework of the RADON[14] project.

S. Dalla Palma et al. [31] notes that in most IaC datasets defective scripts are considerably outnumbered by correct scripts. This may be an issue for classification approaches based on machine learning, because they need training datasets

[12]https://github.com/SODALITE-EU/defect-prediction

[13]https://sodalite.eu/

[14]https://radon-h2020.eu/

TABLE I
CODE SMELLS IN IAC

Smell Name(s)	Description	Cat.	Platforms	Ref.	Tools
Admin by Default	Access to a resource is obtained through a user with excessive privileges. E.g., a database is accessed by the admin user.	–	Chef, Puppet, TOSCA	[11], [15], [16]	SLIC, SLAC, SODALITE
Avoid Comments	The script contains comments.	I, A	Chef	[10]	Foodcritic†
Broken Hierarchy	Inheritance is not used within the same module. E.g., a resource inherits from one in a different namespace.	D	Puppet	[9]	Puppeteer
Complex Expression	The script contains a long and convoluted expression.	I	Puppet	[9]	Puppeteer
Deficient Encapsulation	A class or module has too many global variables referenced by other classes.	D	Puppet	[9]	Puppeteer
Dense Structure	The overall infrastructure has a dense dependency graph.	D	Puppet	[9]	Puppeteer
Deprecated statement usage	The script uses statements deprecated by the platform maintainers.	I	Puppet	[9]	Puppeteer
Duplicate Block	The script contains a number of consecutive duplicate lines higher than a threshold.	I, A	Chef, Puppet	[9], [10]	Puppeteer, Foodcritic†
Duplicate Entry	Duplicate hard-coded parameters or property values.	I	Puppet	[9]	Puppeteer
Empty Default	A Chef project lacks a `default.rb` file or it is empty.	D, S	Chef	[10]	Foodcritic†
Empty Password	The empty string is used as a password.	–	Ansible, Puppet, TOSCA	[11], [15], [16]	SLIC, SLAC, SODALITE
Hard-coded Secret	Sensitive data (e.g., user names, passwords, SSH keys, etc) are hard-coded into the script.	–	Ansible, Chef, Puppet, TOSCA	[11], [15], [16]	SLIC, SLAC, SODALITE
Hyphens	The Chef style guide discourages hyphens in cookbook names.	I, S	Chef	[10]	Foodcritic†
Imperative Abstraction	Imperative statements are used in a declarative language.	D	Puppet	[9]	Puppeteer
Improper Alignment	Code indentation is inconsistent, or contains tabs.	I, A	Chef, Puppet	[9], [10]	Puppeteer, Foodcritic†
Improper Quote Usage	Single and double quotes are used when they should not, or are not used when they should.	I, E	Chef, Puppet	[9], [10]	Puppeteer, Foodcritic†
Include Consistency	The project contains transitive dependencies between similar modules. E.g., a module A references a module B, and they both reference two modules offering similar features.	D, E	Chef	[10]	Foodcritic†
Incomplete Conditional	An `if` statement lacks an `else` clause.	I	Puppet	[9]	Puppeteer
Incomplete tasks, Suspicious comments	The code contains "TODO" or "FIXME" comments.	I	Ansible, Chef, Puppet, TOSCA	[9], [11], [15], [16]	Puppeteer, SLIC, SLAC, SODALITE
Inconsistent naming convention	Naming conventions used in the script deviate from the conventional ones.	I	Puppet, TOSCA	[9], [16]	Puppeteer, SODALITE
Insufficient Key Size	A cryptography key is smaller than a threshold size.	–	TOSCA	[16]	SODALITE
Insufficient modularization	The size of a class or module is excessive (above a certain threshold).	D, E	Chef, Puppet	[9], [10]	Puppeteer, Foodcritic†
Invalid IP address binding, Unrestricted IP address	A resource is assigned the IP address `0.0.0.0`.	–	Ansible, Chef, Puppet, TOSCA	[11], [15], [16]	SLIC, SLAC, SODALITE
Invalid Port Ranges	TCP port numbers are not between 0 and 65535.	–	TOSCA	[16]	SODALITE
Invalid Property Value	Property or attribute has forbidden value. E.g., malformed file mode mask.	I	Puppet	[9]	Puppeteer
Law of Demeter	The project has transitive dependencies. E.g., a module A references a module B, and they both reference a module C.	D, E	Chef	[10]	Foodcritic†
Long Resource	A resource definition spans an excessive number of lines.	D, A	Chef	[10]	Foodcritic†
Long Statement	The script contains long lines (that do not fit in a screen).	I, A	Chef, Puppet	[9], [10]	Puppeteer, Foodcritic†
Misplaced Attribute	Attributes or properties are sorted differently from the conventional order.	I, A	Chef, Puppet	[9], [10]	Puppeteer, Foodcritic†
Missing Abstraction	Resources are not encapsulated in appropriate abstractions.	D	Puppet	[9]	Puppeteer
Missing Default Case	Default case missing in a `switch`, `case` or `selector` statement.	I	Chef, Puppet	[9], [15]	Puppeteer, SLAC
Multifaceted Abstraction	An abstraction violates the *single responsibility principle* [19]. E.g., a resource declaration corresponds to more than one physical resource, or elements declared in a module are not cohesive.	D, A	Chef, Puppet	[9], [10]	Puppeteer, Foodcritic†
No Integrity Check	A file is downloaded without checking it for integrity with, e.g., checksums or GPG signatures.	–	Ansible, Chef	[15]	SLAC
Too many Attributes	A resource has to many attributes (above a threshold).	D, A	Chef	[10]	Foodcritic†
Unguarded Variable	A variable is not enclosed in braces when used in a string.	I, A	Chef, Puppet	[9], [10]	Puppeteer, Foodcritic†
Unstructured Module	A module is not structured in the conventional way.	D, E	Chef, Puppet	[9], [10]	Puppeteer, Foodcritic†
Unnecessary Abstraction	The script contains an empty module or class.	D	Puppet	[9]	Puppeteer
Use of HTTP without TLS, Insecure Communication	Transport Layer Security is not used by default (HTTP instead of HTTPS).	–	Ansible, Chef, Puppet, TOSCA	[11], [15], [16]	SLIC, SLAC, SODALITE
Use of weak cryptography algorithms	Deprecated algorithms are used for encryption (e.g., MD4, MD5, SHA1).	–	Chef, Puppet, TOSCA	[11], [15], [16]	SLIC, SLAC, SODALITE
Weakened Modularity	A module has a higher proportion of inter-module (i.e., external) references than intra-module (i.e., internal).	D, E	Chef, Puppet	[9], [10]	Puppeteer, Foodcritic†

that contain a sufficiently representative variety of samples in both classes. Thus, they employ machine-learning techniques for *novelty detection*, which are trained on a dataset of purely correct scripts, and detect defects by finding scripts with features that deviate from the training dataset. They apply the techniques *OneClassSVN*, *LocalOutlierFactor*, and *IsolationForest* [32] to the same dataset of [27] through 10-fold cross-validation, using only correct scripts for training. The empirical evaluation yields a mean precision ranging from 0.84 (OneClassSVM) to 0.86 (LocalOutlierFactor and IsolationForest), and a mean recall from 0.70 (LocalOutlierFactor) to 0.77 (OneClassSVM and IsolationForest). All approaches greatly outperform *RandomForest* [8].

C. Other

We describe here works that do not fall in one of the two previous categories.

T. Dai et al. [33] developed a tool called SecureCode, which checks shell scripts embedded in or invoked by Ansible scripts. Although the object of validation are shell scripts and not IaC directly, we include this paper in the review because it highlights the fact that IaC verification tools may need to cope with the fact that IaC often relies on external resources that may present issues too. SecureCode scans IaC scripts for references to shell scripts, and when it finds shell script templates, it produces concrete scripts for them by instantiating Ansible variables in them. Then, the scripts are fed to the existing shell-script static analysis tools ShellCheck[15] and PSScriptAnalyzer[16]. SecureCode classifies detected issues based on whether they affect security, availability, performance, or reliability. The authors perform an empirical study on 1,492 scripts from 45 GitHub repositories, on which SecureCode detects 3,535 issues, 116 of which are false positives. They do not estimate precision and recall.

Sommelier[17], by A. Brogi et al. [34], is a tool aimed at validating relationships between nodes in TOSCA service templates. The TOSCA standard prescribes that all elements referenced by node relationships must exist in the service template. Service templates containing undefined references may lead to errors during deployments, if they are not checked. Thus, the authors give formal definitions of such requirements, and describe conditions in which they are violated, thus allowing for *ad hoc* checks for their validity.

V. Model checking of IaC

Model checking [35] is a formal-verification approach where an engineered system is modeled in a logical framework in which it is feasible to check that certain desired properties are guaranteed, or whether undesirable situations may occur. Traditionally, systems are modeled with some kind of graph or transition system, which are natural representations for evolving stateful systems such as electronic devices or imperative programs. Recently, techniques involving SAT (*Boolean Satisfiability*) and SMT (*Satisfiability Modulo Theory*) solvers appeared. They leverage advancements in algorithms for SAT (such as DPLL) to provide tools that often are very fast on practical models, despite the theoretical complexity bounds.

The translation of the verification target into the modeling language can be performed manually by a software architect, or automatically. Manual translation benefits from deep knowledge of the target, but it can be laborious and error-prone. Devising an automatic translator is seldom trivial, and often comes with a cost in generality.

Due to well-known results in Computer Science, there is no automatic procedure capable of singling out all and only the pieces of software, written in a Turing-complete programming language, whose behavior conforms to a given non-trivial property. Consequently, automatic verification approaches must be more restrictive than it would be desirable in guaranteeing that a program satisfies a requirement. This does not necessarily apply to IaC, where the concern is rather about the capability of the logical framework to express system properties. E.g., a model focused on relationships between cloud computing nodes will not be able to predict an internal application malfunction on one of such nodes.

In this section, we describe works in IaC model checking.

K. Jayaraman et al. [36] developed SecGuru, a tool that analyses firewall ACLs within infrastructures deployed in the Azure cloud. SecGuru enables inspection of the differences resulting from an ACL update in terms of network packets that are allowed or blocked by the firewall, and to check the behavior of the firewall against a given *policy*. This is obtained by encoding the action taken by the firewall on packets and the policies into an SMT problem, then solved by the Z3 SMT tool. The resulting instances correspond to the network packets that are either blocked or allowed. The tool was evaluated on real and synthetic policies, and is currently active in the Azure cloud, reportedly having a "measurable positive impact in prohibiting policy misconfigurations".

A. Brogi et al. [37] propose a tool to verify the validity of TOSCA management plans. The user enriches each TOSCA node template with a set compatible states. Then, they characterize management operations and states by specifying the states in which the nodes providing the required capabilities need to be for the execution of the plan to be successful. The tool checks the validity of a plan by creating a state-representation of the whole infrastructure. In this representation, a state is a valid combination of the states of individual nodes, and a transition between states is a management operation on a node such that the operation's requirements are satisfied. The validity of the plan is then equivalent to the existence of an operation-labeled path.

W. Chareonsuk and W. Vatanawood [40] present a toolchain to perform formal verification of interacting web services in a TOSCA specification. The process relies on user-supplied information describing the behavior of the modelled services, written in the *Web Services Business Process Execution Language* (WS-BPEL, or BPEL). The user provides a BPEL description of services running on the infrastructure, and

[15] https://www.shellcheck.net/

[16] https://docs.microsoft.com/en-us/powershell/module/psscriptanalyzer/

[17] https://github.com/di-unipi-socc/Sommelier

TABLE II
ANSWERS TO THE RQS

Tool name (RQ1)	Ref.	Target Platform (RQ4)	Target Properties (RQ3)	Technique (RQ2)
ACID	[18]	Puppet	Anti-patterns	Pattern rules
Barrel	[37]	TOSCA	Deployment plan correctness	Reachability in Transition Systems
DeepIaC	[25]	Ansible	Anti-patterns	Deep Learning
Foodcritic with custom rules	[10]	Chef	Code smells	Pattern rules and code metrics
Häyhä	[38]	CloudFormation	Security vulnerabilities	Dataflow graph analysis
Puppeteer	[9]	Puppet	Code smells	Pattern rules and code metrics
RADON defect prediction framework	[27]	Ansible	Anti-patterns	Data Mining
Rehearsal	[39]	Puppet	Determinism and idempotency	SMT solving
SecGuru	[36]	Azure ACL	Network policies	SMT solving
SecureCode	[33]	Ansible	Bugs in shell scripts	External tools
SLIC	[11]	Puppet	Code smells	Pattern rules
SLAC	[15]	Ansible, Chef	Code smells	Pattern rules
SODALITE defect predictor	[16]	TOSCA	Code smells	SPARQL rules on OWL2 ontology
Sommelier	[34]	TOSCA	Undefined references	*Ad hoc* algorithms
–	[40]	TOSCA	LTL on deployed services behavior	Model Checking (SPIN)
–	[31]	Ansible	Anti-patterns	Data Mining
–	[20]	Puppet	Anti-patterns	Data Mining
–	[24]	Puppet	Anti-patterns	Data Mining
–	[41]	TOSCA	Success of orchestrated operations	Interactive Theorem Proving

integrates it in the TOSCA service template using *ad hoc* node and relationship types. This IaC is then compiled into a PROMELA specification, against which *Linear Temporal Logic* (LTL) [35] formulas can be verified using the SPIN model checker [42]. As an example, the authors propose checking *safety* properties, expressed as formulas of the form $\Box\neg P$, i.e. "it is always true that P does not hold", where P is some undesirable condition.

In [39], R. Shambaugh et al. developed Rehearsal, a tool to analyze Puppet configurations by compiling them into a formal language describing filesystem operations, and then translating them into SMT specifications. The Z3 SMT solver is used to look for models representing executions that violate the principles of determinism and idempotency. Rehearsal was tested on a small set of 13 Puppet configurations gathered from GitHub and Puppet Forge, and found bugs in 6 of them. Benchmarks on the test set run on a quad-core 3.5 GHz Intel Core i5 with 8GB RAM measured average checking times within 3 seconds for all configurations.

J. Lepiller et al. [38] identified a class of cloud-related security vulnerabilities, called *intra-update sniping vulnerabilities*. These occur when an infrastructure update operation, despite transitioning between secure states, traverses insecure intermediate states, for instance because components are updated in the wrong order. To detect this vulnerability class in CloudFormation templates, the authors developed Häyhä, which models the described infrastructure as a *dataflow graph*. Häyhä was evaluated on a set of open-source CloudFormation templates: while no vulnerability was detected, the tool showed performances acceptable for integration in a deployment workflow, as execution time was within 1 second for all templates.

H. Yoshida et al. [41] propose a method for manual modeling of TOSCA service templates in the formal specification language CafeOBJ. This method is useful for proving that orchestration operations can reach the final state of the infrastructure while maintaining a given invariant property in intermediate states. Proving a property modeled in CafeOBJ is a form of formal verification, but it cannot be regarded as model checking since it requires user interaction.

A number of authors presented techniques for model checking of cloud infrastructure in which the model is constructed manually. H. Sahli et al. [43] proposed *bigraphical reactive systems* as a suitable logical framework to model cloud infrastructure lifecycles, and exemplified the use of the model checker BigMC to check relevant properties of elasticity and plasticity. K. Klai and H. Ochi [44] presented a technique for model checking the interaction of multiple cloud services accessing shared resources concurrently. The cloud services are modeled as *RCoWF* (*Resource-Constraint open WorkFlow*) nets and translated into labeled Kripke structures, against which *hybrid LTL* formulae are checked, e.g. to detect deadlocks on a concurrently accessed resource. We do not describe these works further, because they do not address IaC directly.

VI. DISCUSSION AND CONCLUSIONS

We conducted a literature review on IaC static analysis techniques by querying the most important bibliographic search engines. We summarize the answers to our research questions in Table II.

Concerning RQ2, we found out that the most used techniques powering the tools are string-pattern rules (5 tools), and machine learning techniques (5 tools). Model checking is used in 5 tools too, with SMT solvers being most popular.

Regarding RQ3, *anti-patterns*, by which we mean code or process metrics and other features used for training machine learning classifiers, and code smells are the most targeted properties (resp. 6 and 5 tools). Model-checking-based tools often target the runtime behavior of the deployment.

As for RQ4, the most targeted platform is Puppet (6 tools), followed by Ansible (4 tools) and TOSCA (4 tools).

In conclusion, the review shows an increasing attention on quality assurance techniques for IaC. Code smell and defect detection and prediction techniques have reached considerable advancement, although improvements may be possible in

terms of accuracy. Other future-work lines are investigation of automated remediation strategies for defects, and use of techniques that take the IaC semantics into account, as opposed to currently used pattern-matching and machine-learning.

A direction for future work in model checking tools is to increase the precision of abstractions used for modeling deployments, to enable the verification of more properties. Better automation of IaC modeling should also be targeted, because several works still employ manual modeling, which is impractical and error-prone.

REFERENCES

[1] K. Morris, *Infrastructure as Code*. O'Reilly Media, 2016.

[2] L. J. Bass, I. M. Weber, and L. Zhu, *DevOps - A Software Architect's Perspective*. Addison-Wesley, 2015.

[3] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code - an empirical study," in *Proc. 12th Work. Conf. Mining Softw. Repositories, MSR'15*. IEEE Comput. Soc., 2015, pp. 45–55.

[4] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Inf. Softw. Technol.*, vol. 108, pp. 65–77, 2019.

[5] A. Alnafessah, A. U. Gias, R. Wang, L. Zhu, G. Casale, and A. Filieri, "Quality-aware devops research: Where do we stand?" *IEEE Access*, vol. 9, pp. 44 476–44 489, 2021.

[6] O. Open, "TOSCA simple profile in YAML version 1.3," 2019.

[7] M. Fowler, *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] P. Tan, M. S. Steinbach, A. Karpatne, and V. Kumar, *Introduction to Data Mining (Second Edition)*. Pearson, 2019.

[9] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proc. 13th Int. Conf. Mining Softw. Repositories, MSR'16*. ACM, 2016, pp. 189–200.

[10] J. Schwarz, A. Steffens, and H. Lichter, "Code smells in infrastructure as code," in *Proc. 11th Int. Conf. Qual. Inf. Commun. Technol., QUATIC'18*. IEEE Comp. Soc., 2018, pp. 220–228.

[11] A. Rahman, C. Parnin, and L. A. Williams, "The seven sins: security smells in infrastructure as code scripts," in *Proc. 41st Int. Conf. Softw. Eng., ICSE'19*. IEEE/ACM, 2019, pp. 164–175.

[12] J. Saldana, *The coding manual for qualitative researchers*. SAGE, 2015.

[13] "Common Weakness Enumeration," The MITRE Corporation. [Online]. Available: https://cwe.mitre.org/

[14] F. A. Bhuiyan and A. Rahman, "Characterizing co-located insecure coding patterns in infrastructure as code scripts," in *Proc. 35th Int. Conf. Autom. Softw. Eng. Workshops, ASEW'20*. ACM, 2020, pp. 27–32.

[15] A. Rahman, M. R. Rahman, C. Parnin, and L. A. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, pp. 3:1–3:31, 2021.

[16] I. Kumara, Z. Vasileiou, G. Meditskos, D. A. Tamburri, W. van den Heuvel, A. Karakostas, S. Vrochidis, and I. Kompatsiaris, "Towards semantic detection of smells in cloud infrastructure code," in *Proc. 10th Int. Conf. Web Intell., Mining and Semantics, WIMS'20*. ACM, 2020, pp. 63–67.

[17] "SPARQL 1.1 query language," W3C, 2013. [Online]. Available: http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[18] A. Rahman, E. Farhana, C. Parnin, and L. A. Williams, "Gang of eight: a defect taxonomy for infrastructure as code scripts," in *Proc. 42nd Int. Conf. Softw. Eng., ICSE'20*. ACM, 2020, pp. 752–764.

[19] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 2003.

[20] A. Rahman and L. A. Williams, "Characterizing defective configuration scripts used for continuous deployment," in *Proc. 11th IEEE Int. Conf. Softw. Testing, Verification Validation, ICST'18*. IEEE Comp. Soc., 2018, pp. 34–45.

[21] Z. S. Harris, "Distributional structure," *WORD*, vol. 10, no. 2-3, pp. 146–162, 1954.

[22] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge Univ. Press, 2008.

[23] J. Corbin and A. Strauss, *Basics of Qualitative Research*. SAGE, 2015.

[24] A. Rahman and L. A. Williams, "Source code properties of defective infrastructure as code scripts," *Inf. Softw. Technol.*, vol. 112, pp. 148–163, 2019.

[25] N. Borovits, I. Kumara, P. Krishnan, S. D. Palma, D. D. Nucci, F. Palomba, D. A. Tamburri, and W. van den Heuvel, "DeepIaC: deep learning-based linguistic anti-pattern detection in IaC," in *Proc. 4th Int. Workshop Mach. Learn. Techn. Softw. Qual. Eval., MaLTeSQuE'20*. ACM, 2020, pp. 7–12.

[26] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[27] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Within-project defect prediction of infrastructure-as-code using product and process metrics," *IEEE Trans. Softw. Eng.*, pp. 1–1, 2021, to appear. [Online]. Available: https://doi.org/10.1109/TSE.2021.3051492

[28] S. Dalla Palma, "Defect prediction tool validation dataset 2," Zenodo, 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4299908

[29] F. Rahman and P. T. Devanbu, "How, and why, process metrics are better," in *Proc. 35th Int. Conf. Softw. Eng., ICSE'13*. IEEE Comp. Soc., 2013, pp. 432–441.

[30] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Toward a catalog of software quality metrics for infrastructure code," *J. Syst. Softw.*, vol. 170, p. 110726, 2020.

[31] S. D. Palma, M. Mohammadi, D. D. Nucci, and D. A. Tamburri, "Singling the odd ones out: a novelty detection approach to find defects in infrastructure-as-code," in *Proc. 4th Int. Workshop Mach. Learn. Techn. Softw. Qual. Eval., MaLTeSQuE'20*. ACM, 2020, pp. 31–36.

[32] M. Markou and S. Singh, "Novelty detection: a review - part 1: statistical approaches," *Signal Process.*, vol. 83, no. 12, pp. 2481–2497, 2003.

[33] T. Dai, A. A. Karve, G. Koper, and S. Zeng, "Automatically detecting risky scripts in infrastructure code," in *Proc. ACM Symp. Cloud Comput., SoCC'20*. ACM, 2020, pp. 358–371.

[34] A. Brogi, A. Di Tommaso, and J. Soldani, "Sommelier: A tool for validating TOSCA application topologies," in *Proc. 5th Int. Conf. Model-Driven Eng. and Softw. Develop., MODELSWARD'17*, ser. Commun. Comput. Inf. Sci., vol. 880. Springer, 2017, pp. 1–22.

[35] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer, 2018.

[36] K. Jayaraman, N. Bjørner, G. Outhred, and C. Kaufman, "Automated analysis and debugging of network connectivity policies," Microsoft, Tech. Rep. MSR-TR-2014-102, 2014. [Online]. Available: https://www.microsoft.com/en-us/research/publication/automated-analysis-and-debugging-of-network-connectivity-policies/

[37] A. Brogi, A. Canciani, and J. Soldani, "Modelling and analysing cloud application management," in *Proc. 4th Eur. Conf. Service Oriented Cloud Comput. ESOCC'15*, ser. LNCS, vol. 9306. Springer, 2015, pp. 19–33.

[38] J. Lepiller, R. Piskac, M. Schäf, and M. Santolucito, "Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities," in *Proc. 27th Int. Conf. Tools Alg. for the Constr. and Anal. of Syst., TACAS'21, Part II*, ser. LNCS, vol. 12652. Springer, 2021, pp. 105–123.

[39] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: a configuration verification tool for puppet," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Impl., PLDI'16*. ACM, 2016, pp. 416–430.

[40] W. Chareonsuk and W. Vatanawood, "Formal verification of cloud orchestration design with TOSCA and BPEL," in *Proc. 13th Int. Conf. Elect. Eng./Electron., Comput., Telecomm. Inf. Technol., ECTI-CON'16*. IEEE, 2016, pp. 1–5.

[41] H. Yoshida, K. Ogata, and K. Futatsugi, "Formalization and verification of declarative cloud orchestration," in *Proc. 17th Int. Conf. Formal Methods Softw. Eng., ICFEM'15*, ser. LNCS, vol. 9407. Springer, 2015, pp. 33–49.

[42] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[43] H. Sahli, F. Belala, and C. Bouanaka, "Model-checking cloud systems using BigMC," in *Proc. 8th Int. Workshop Verification Eval. Comput. Commun. Syst., VECoS'14*, ser. CEUR Workshop Proc., vol. 1256. CEUR-WS.org, 2014, pp. 25–33.

[44] K. Klai and H. Ochi, "Model checking of composite cloud services," in *Proc. IEEE Int. Conf. Web Services, ICWS'16*. IEEE Comp. Soc., 2016, pp. 356–363.