ORIGINAL ARTICLE

"Through the looking-glass..." An Empirical Study on Blob Infrastructure Blueprints in TOSCA

Stefano Dalla Palma[1] | Chiel van Asseldonk[1] | Gemma Catolino[1] | Dario Di Nucci[2] | Fabio Palomba[2] | Damian A. Tamburri[3]

[1]Tilburg University, Jheronimous Academy of Data Science, Netherlands

[2]Software Engineering (SeSa) Lab — University of Salerno, Italy

[3]Eindhoven University of Technology, Jheronimous Academy of Data Science, Netherlands

Correspondence
Stefano Dalla Palma, Jheronimus Academy of Data Science, Tilburg University, City, The Netherlands
Email: s.dallapalma@uvt.nl

Funding information

Infrastructure-as-Code (IaC) helps keep up with the demand for fast, reliable, high-quality services by provisioning and managing infrastructures through configuration files. Those files ensure efficient and repeatable routines for system provisioning, but they might be affected by *code smells* that negatively affect quality and code maintenance.

Research has broadly studied code smells for traditional source code development; however, none explored them in the "Topology and Orchestration Specification for Cloud Applications" (TOSCA), the technology-agnostic OASIS standard for IaC. In this paper, we investigate a prominent traditional implementation code smell potentially applicable to TOSCA: *Large Class*, or "Blob Blueprint" in IaC terms.

We compare metrics-based and unsupervised learning-based detectors on a large dataset of manually validated observations related to *Blob Blueprints*. We provide insights on code metrics that corroborate previous findings and empirically show that metrics-based detectors perform highly in detecting *Blob Blueprints*.

We deem our results put forward a new research path toward dealing with this problem, e.g., in the scope of fully automated service pipelines.

KEYWORDS
Infrastructure-as-Code, Code Smells, Metrics-based detectors, Unsupervised Learning, TOSCA

1 | INTRODUCTION

Since the rise of DevOps [1], the shift from on-premise infrastructure to cloud-based infrastructure has introduced new challenges and opportunities driven by the continuously increasing demand for fast and high-quality software services and their orchestration. DevOps covers a broad spectrum of organizational and technical practices to achieve this integration, including Continuous Integration (CI) and Continuous Deployment (CD). In particular, service orchestration is strongly influenced by *Infrastructure-as-Code (IaC)*, which rapidly became a crucial practice to automate infrastructure and ensure consistent and repeatable routines for service provisioning and configuration changes [2].

On the one hand, IaC substantially benefits the maintainability and quality of the overall service properties and service-level agreements, e.g., faster closing-the-loop iterations and, therefore, faster service evolution cycles. At the same time, it reduces the time, effort, and specialized skills required to provision and scale infrastructure services while improving consistency by reducing ad-hoc configuration changes and updates (a phenomenon known as configuration drift[1]). Nevertheless, on the other hand, IaC is still code and, therefore, subject to all potential problems (e.g., due to bad coding practices or human error). Similarly to software written in traditional application languages, IaC artifacts – often called *blueprints* – bear the same coding horror[2] fatalities, which can become even more impactful for IaC, where bad coding practices contribute to introducing IaC defects [3]; their impact can be massive since it manifests at runtime and often via running expensive infrastructure costs and connected shortfalls. In 2017, for example, a services infrastructure failure within Amazon Web Services took down websites such as *Expedia.com*, *Slack.com*, *Medium.com*, and the US Securities and Exchange Commission for several hours.[3]

In this study, we focus on bad coding practices, which reveal another harm in software development: symptoms indicating wrong style usage or a lousy design known as ***code smells***. Smells do not directly cause system failures but violate best practices and design principles, negatively affecting readability and code maintainability [4].

Motivation. Code smells are broadly researched for traditional source code development [5], as their detection can enhance software quality. In the scope of IaC, Guerriero et al. [6] investigated the state of practices in adopting IaC using the data from 44 semi-structured interviews with senior developers. They observed that large IaC scripts, referred to as ***Blob Blueprint***, occur among the most common bad practices in the industry when developing infrastructure code.

Unfortunately, only a few works exist on IaC code smells [7], and they focus on specific technologies and languages, such as Puppet or Ansible. However, Guerriero et al. [6] identify as one of the best practices "recombining diverse formats by abstraction using the OASIS TOSCA standard for IaC and including multiple formats inside node-type definitions". Indeed, technology-agnostic infrastructure code, such as the OASIS Topology and Orchestration Specification for Cloud Applications[4] (TOSCA) can build upon existing configuration and orchestration languages to improve the readability and portability of configuration files across platforms [8, 9]. It enables automated deployment of technology-independent and multi-cloud compliant applications, managing applications, resources, and services regardless of the underlying cloud platform or infrastructure. From a business viewpoint, TOSCA *"expands customer*

[1] https://www.ibm.com/cloud/learn/infrastructure-as-code

[2] https://blog.codinghorror.com/please-dont-learn-to-code/

[3] https://aws.amazon.com/message/41926/

[4] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca

choice, *reduces cost*, and *increases business agility* across the application life cycle. The synergy between these benefits *accelerates overall time-to-value"* [9].

What is more, the fourth and sixth authors of the paper are active members of the OASIS TOSCA Standard Technical Committee and used this presence to enact discussions about this work in some of the committee's meetings. As a result, practitioners shared concerns about "long, blob-like blueprints". In particular, they mentioned that the most critical hazards connected to infrastructure abuse come from lousy coding practices concerning the security of IaC blueprints. The targets for such practices are mainly found in long, *blob-like* blueprints. Such issues reportedly can yield irreparable infrastructure damage as well as loss or even theft of data as much as leaking of industrial secrets to a point in which manual inspection of long blueprints is required.[5]

Research Questions. In this study, we conjecture that identifying code smells in technology-agnostic infrastructure code (i.e., TOSCA) and related metrics opens up opportunities for building a general, automated service continuity quality model designed explicitly for configuration orchestration languages. Furthermore, we deem that analyzing such code smells paves the way to understanding how lousy coding practices affect service infrastructure continuity [10]. In particular, the goal is to identify structural code measures that characterize complex blueprints and analyze the effectiveness of candidate metric- and unsupervised learning-based techniques in detecting those blueprints. Motivated by this goal, this study aims at addressing the following research questions in the context of TOSCA:

RQ$_1$ *To what extent can structural code metrics distinguish between Blob and sound blueprints?*
RQ$_2$ *To what extent can metric- and unsupervised learning-based techniques detect Blob Blueprint?*
RQ$_3$ *What metrics are the most effective to maximize the performance of those detectors?*

We conducted a case study involving 749 blueprints and a prominent traditional implementation code smell, known as *Large Class* or *Blob*, that highly impacts fault- and change-proneness [11, 12, 13] and that could potentially be observed and easily implemented in TOSCA. The smell is one of the most frequently investigated for traditional application code [14]. In addition, Guerriero et al. [6] observed it among the most common bad practices in the industry when developing infrastructure code; they refer to it as ***Blob Blueprint***, namely a too-large IaC script. From here on, we use the same nomenclature. We selected this smell for its implementation ease, frequency, and potential impact on infrastructure code quality. We build upon the studies by Sharma et al. [15] and Schwarz et al. [16]. Specifically, we analyze traditional structural code metrics for IaC smell detection to corroborate their findings on a technology-agnostic language (i.e., TOSCA). Additionally, and investigate how metric- and unsupervised learning-based techniques perform to detect Blob Blueprints.

The motivation behind focusing on the metrics- and unsupervised learning-based detectors is two-fold. On the one hand, metrics-based smell detectors are the most frequent and easy to implement [17]. They calculate a set of metrics, such as *lines of code*, *coupling*, and *cohesion*, upon the original source code and detect smells if they exceed a given threshold [17]. However, determining a suitable threshold demanded by metrics-based smell detectors is a non-trivial challenge. On the other hand, the interest in ML-based methods, prevalently supervised, is growing to overcome shortcomings such as determining threshold values. However, they come with other drawbacks, such as the need for accurate (labeled) training data, which might be hard to acquire [18]. Despite this, unsupervised learning might significantly reduce the effort of collecting and identifying smelly blueprints, similarly to previous work on software defect prediction [19, 20].

[5] https://www.oasis-open.org/2022/06/06/emerging-compute-models-recommendations-and-sample-profile-v1-0-published-by-tosca-tc/ (Section 2.1)

Contribution. This study contributes to research with an empirical study that compares metric- and unsupervised learning-based techniques to detect Blob Blueprint. In particular, we compare several popular clustering techniques with a detector applying the Interquartile Rule on a dataset of manually validated observations related to *Blob Blueprints*. We provide insights on code metrics that corroborate previous findings and empirically show that metric-based detectors perform well in detecting *Blob Blueprints*. Finally, we provide a replication package and a comprehensive dataset of publicly available TOSCA blueprints, including source code measurements calculated on these blueprints and manually validated observations related to the *Blob Blueprint* smell.[6]

Paper Structure. Section 2 presents background on Infrastructure-as-Code, focusing on TOSCA, and reviews the existing literature on IaC smell detection. Section 3 outlines the measures for blueprint complexity that characterize Blob Blueprints. Section 4 describes the empirical study to evaluate metric- and unsupervised learning-based techniques in detecting Blob Blueprints, and Section 5 reports the experiment results. Section 6 discusses limitations and threats to validity; Section 7 discusses the implications for researchers and practitioners and lessons learned. Section 8 concludes the paper and outlines future work.

2 | BACKGROUND AND RELATED WORK

This section provides a brief grounding about Infrastructure-as-Code (IaC), and TOSCA, as well as previous literature on code smells in IaC.

2.1 | IaC and TOSCA: An Overview

The Topology and Orchestration Specification for Cloud Applications (TOSCA) is the official OASIS industry standard for IaC. It is a YAML-based domain-specific language that allows for the automated deployment of technology-independent and multi-cloud compliant applications. In other words, it can manage applications, resources, and services regardless of the underlying cloud platform, software environment, or infrastructure [9]. Furthermore, unlike other configuration management tools, such as Puppet, Chef, Docker, and Ansible, TOSCA covers the complete application life cycle rather than just deployment and configuration management [9]. Thus, it provides a higher abstraction level while incorporating those above and additional technologies to serve a specific need.

The creator of a cloud service captures its structure in a service topology – a graph of *nodes* representing the service's components and *relationships* that connect and structure nodes into the topology. Both nodes and relationships are typed and hold a set of type-specific properties. *Types* define reusable entities that define the semantics of the node or relationship (e.g., properties, attributes, requirements, capabilities, interfaces). *Templates* form the cloud service's topology using these types. In particular, they define how to instantiate the respective type for use in the application. They allow defining the start values of the properties by specifying their defaults. However, they can overwrite and extend the types to adjust them for their respective application. These types are conceptually comparable to abstract classes in Java, whereas the templates are comparable to concrete classes [8].

[6] `https://github.com/jade-lab/tosca-smells`

3 | THEORETICAL MODEL: THE BLOB BLUEPRINT

With this work, we do not intend to introduce new smells for IaC, but we conduct a case study revolving around one specific code smell called *Blob Blueprint*. In traditional application code, researchers refer to this smell as *Large Class*; it represents a class that typically contains too many fields and methods and relies on several external data classes, making it low cohesive [4, 5].

In IaC, only two works relate to Blob Blueprint, although they do not target it directly [15, 16]. For example, Sharma et al. [15] and Schwarz et al. [16] presented *Insufficient Modularization*. This smell represents an abstraction (e.g., a resource, class, "define", or module) that is large or complex and thus can be modularized further. They instantiated it for Puppet and Chef, respectively, and provided three conditions for their detection:

1. configuration files that contain more than one class (in Chef) or resource (in Puppet); or
2. class declarations that are too large (more than 40 lines of code); or
3. class declarations that are too complex (max nesting depth more than three).

In TOSCA, node and relationship *types* and *templates* are analogous to abstract and concrete classes [8]. Therefore, their size is a leading indicator for *Blob Blueprints*, and we consider the cumulative number of types and templates in condition (1). As for condition (2), it is accepted that the larger a module, the more difficult it is to comprehend. Indeed, on average, the number of conditions can likely increase proportionally to the module size.[7] Hence, we consider the number of code lines as a simple measure of its size. Finally, condition (3) is computed in the respective paper using the maximum nesting depth (e.g., in an *if*) for an abstraction. Unfortunately, this is impossible in TOSCA because of its declarative nature, and we had to define a different complexity measure. However, because of its novelty and difference compared with traditional programming languages, it is unclear what can be considered a *complexity* measure in TOSCA.

In general, a complexity measure tries to capture the difficulty in understanding a module (i.e., a blueprint in this case). Following the definition of Large Class above, we computed the number of *interfaces* and *properties* as analogous to the number of methods and attributes. In addition, in line with previous studies on the matter [15], we further computed the well-known *lack of cohesion of methods* (LCOM) [21] since a higher value of LCOM indicates decreased encapsulation and increased complexity [22, 23]. In particular, the latter measures the number of connected components in a class, where a connected component is a set of related methods and class-level variables. First, related methods, which access the same class-level variables, are grouped. Then, LCOM equals the number of connected groups of methods. Ideally, there should be only one component in each class. Unfortunately, due to its different structure and characteristics, we cannot use the same metric for infrastructure code. Thus, our model defines a *connected component* as a set of related types or templates (rather than methods in traditional languages) and blueprint-level properties (rather than attributes in traditional languages). Then, similarly to Sharma et al. [15], we use the following algorithm to compute LCOM in a blueprint:

1. Consider each declared element, such as node and relationship templates, as a node in a graph. Initially, the graph contains the disconnected components (DC) equal to the number of elements.
2. Identify the parameters of the topology template and the used variables. We refer to these elements as data members.

[7] `https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html`

```yaml
tosca_definitions_version: "tosca_simple_yaml_1_0"

# Imports, description, and metadata...

topology_template:
  inputs:
    mem_size:
      # ...
    num_cpus:
      # ...
    rclone_user:
      # ...
    rclone_password:
      # ...

  node_templates:
    Docker:
      type: "tosca.nodes.indigo.Docker"
      cababilities:
        host:
          properties:
            mem_size: { get_input: mem_size }
            num_cpus: { get_input: num_cpus }

    marathon:
      type: "tosca.nodes.indigo.Marathon"
      properties:
        command: { get_input: run_command }
        environment_variables:
          RCLONE_CONFIG_USER: { get_input: rclone_user }
          RCLONE_CONFIG_PASS: { get_input: rclone_password }
```

LISTING 1 An example of how the measure LCOM can be interpreted and computed in TOSCA. The two nodes in the topology (*Docker* and *marathon*) access two disjoint groups of input each: *mem_size* and *num_cpu* for *Docker*, and *rclone_user* and *rclone_password* for *marathon*. Therefore, LCOM = 2.

3. For each data member, identify the components that use it and merge the identified components into a single component.
4. Compute the lack of cohesion as $LCOM = |DC|$.

Listing 1 shows an example of this measure for TOSCA. The two topology nodes (*Docker* and *marathon*) access two disjoint groups of input each (i.e., *mem_size* and *num_cpu* for *Docker*, and *rclone_user* and *rclone_password* for *marathon*). Therefore, there are two connected components, each consisting of one node, hence LCOM = 2.

In addition to cohesion, we counted the Number of imports as a measure of the *efferent coupling* (a.k.a. fan-out) that defines the number of components on which a particular component depends. Components with a high efferent coupling value are sensitive to the changes introduced to their dependencies. Moreover, the deficiencies of their dependencies naturally manifest themselves in these components.

Please note that we first relied on our knowledge of "complexity" in application code to elicit a set of relevant metrics in TOSCA. Then, we performed preliminary non-structured interviews with the OASIS TOSCA Technical Committee to understand their view on the matter and evaluate the extent to which our definition of blobs and infrastructure complexity match. We conducted several rounds of interviews, where the experts were free to map IaC characteristics they perceived as factors of complexity to blob code smells. We repeated the process until two key complexity aspects emerged. Such factors still require dedicated attention even beyond the scope of this study:

1. **Blob Blueprints reflect a lower bound for complexity.** Blueprints that implement multiple modules, interface hooks, and dependencies tend to develop maintainability problems and vulnerability to infrastructure penetration or chaos [24].
2. **Automated testing Blob IaC is nigh impossible.** A considerable number of negative characteristics are sparsely related to automated testing (e.g., low code understandability, low code reuse); this warrants the necessity of defining operationally multiple and fine-grained complexity measures to be used in a combined complexity function for further automation.

While this study focuses on the first of the above points, we are releasing all materials and automation borne of this study to encourage further replication of our computational results and further research on the matter. Finally, we refined and implemented the considered metrics based on their input.

4 | STUDY METHODOLOGY

In this section, we present the methodology followed throughout the study, consisting of four phases: (i) *data collection*, (ii) *data preparation* (exploratory analysis and data pre-processing), (iii) *detectors building*, and (iv) *performance evaluation and comparison*. The *goal* is to investigate how metric- and unsupervised learning-based detectors identify Blob Blueprints, to provide improved tooling to identify them in practice. The *perspective* is for researchers and practitioners. The former is interested in assessing, through *in-vitro* experimentation, the effectiveness of metric- and unsupervised learning-based code smell detection applied to TOSCA. The latter is interested in evaluating how unsupervised learning-based smell detection works in practice.

4.1 | Data Collection

TOSCA is a novel standard. To get a comprehensive set of blueprints, we mined GitHub to look for all repositories related to the search query `tosca`. The search returned 636 repositories that we analyzed to collect TOSCA blueprints. First, we discarded repositories with no releases because we are interested in blueprints considered functioning. Then, we collected all the files with the extension *.tosca* from the last release of each project. However, TOSCA blueprints can also have a *.yml* extension. Therefore, we searched for the presence of the keyword *tosca_definitions_version* for YAML files.[8] That keyword identifies the versioned set of normative TOSCA type definitions to validate those types defined in the TOSCA Simple Profile and is mandatory.[9] Please note that we discarded blueprints used for testing or examples, i.e., those containing *test* or *example* in the file path, as they are not representative of production blueprints targeted by this study. This way, we collected 1036 blueprints from 42 repositories.

4.2 | Data Preparation

The blueprints collected in the previous section were scanned to extract the metrics defined in Section 3 to create the dataset for experiments. To this end, we implemented an open-source tool for TOSCA available on GitHub.[10] Please note that 287 blueprints were discarded at this point because of invalid YAML files (123) or duplicates (164) based

[8] This step was performed on Jun 9, 2021.

[9] `https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html`

[10] `https://github.com/radon-h2020/radon-tosca-metrics`

on the extracted metrics, leading to a comprehensive data set of 749 distinct blueprints.

From that, we manually annotated a statistically relevant random set of 290 blueprints that we sampled to have an acceptable margin of error of 5% and a 95% confidence level to create the ground truth for the exploratory analysis described below and compare techniques. The annotation was performed *before* the subsequent analyses so that we could avoid additional bias; furthermore, the inspectors actively discussed their operations multiple times to convey a decision in order to reduce subjectivity.

The first and fourth authors scanned each resource and labeled it as smelly or sound based on their experience and understanding of the blueprint's semantics. The authors have at least *four* years of experience in code quality and IaC research. In addition, the fourth author is an active member of the OASIS TOSCA Technical Committee;[11] as such, he participates in monthly meetings where the Committee discusses with TOSCA practitioners about status and challenges of the language.

Each blueprint was subjectively analyzed considering the overall length, the number of nodes and relationships, and their scope based on type, description, properties, and interfaces; we defined these criteria based on the theoretical model in Section 3 before annotating. We also considered complexity in terms of difficulty in understanding the operations performed by the blueprint. A web application was developed and shared among the assessors to facilitate the annotation.[12]

Then, Cohen's Kappa [25] was measured to compute the degree of agreement between the assessors. Cohen's Kappa ranges between 0 and 1, with 0 indicating no agreement between the two raters and 1 indicating perfect agreement. In case of disagreements, the assessors met and discussed the disagreed blueprint to convey a decision. Following this procedure, we obtained a ground truth consisting of 248 sound and 42 smelly instances after reaching a complete agreement in the resolution phase from an initial Cohen's Kappa of 0.56 (i.e., moderate agreement). Below, we describe the exploratory analysis performed on the ground truth.

4.2.1 | Exploratory Analysis

We tested each metric separately using statistical analysis before employing them for predicting Blob Blueprints. For each metric, we measured whether the distribution of this metric within Blob Blueprints is statistically different from the distribution within all other blueprints. To this end, we applied the non-parametric Mann–Whitney U test [26] with a significance level $\alpha = 0.01$.

To better control for the randomness of our observations, we used Bonferroni's correction [27] to adjust the significance level according to the number of comparisons (i.e., five). Thus, the results are significant at the significance level $\alpha = 0.002$. P-values below this show that the two groups differ for the considered metric. While we acknowledge that a metric distributed differently does not necessarily distinguish Blob and sound blueprints, these results hint at why a machine learning approach that combines these features can be successful.

Beyond the p-value interpretation, we calculated the effect size using Cliff's delta [28] to measure the magnitude of the difference between two populations and ranges from zero to one. For example, according to Kampenes et al. [29], a value below 0.147 is considered trivial; between 0.147 and 0.33, it is small; between 0.33 and 0.474, it is medium, and it is large above 0.474. In the following, we discuss how we pre-processed these metrics for experimentation.

[11] `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca`

[12] Demo accessible at `https://smell-annotator.web.app/` using the token: 9WhBsZe1EiDhFgVXtBPn

4.2.2 | Pre-processing

First, we normalized data as a common requirement for many machine learning estimators. Typically this is done by removing the mean and scaling to unit variance. However, outliers can often influence the sample mean or variance negatively. Therefore, we resorted to the *RobustScaler* available in *scikit-learn* for the task.[13] It scales data similarly to the min-max normalization but uses the interquartile range rather than the max-min range to be robust to outliers. Therefore it follows the formula:

$$\frac{x - Q_1(x)}{Q_3(x) - Q_1(x)}$$

As for feature selection, we meant the metrics for code smell detection. However, some metrics may correlate to others. The latter might be a problem in unsupervised learning, as the concept they represent gets more weight than other concepts. Thus, the final model might skew toward that particular concept, which might be undesirable. For that reason, we controlled for multicollinearity through the Variable Inflation Factor (VIF) [30], discarding the features having a value larger than 10, a widely-used rule-of-thumb [31]. In addition, we used a stepwise forward selection to determine the optimal set of features to build the detectors described below. More specifically, all the metrics but those already selected are tested against the MCC at each step. The metric that significantly improves MCC the most is added to the set.

4.3 | Detectors Building

Afterward, we used the pre-processed data and the selected features to build the metrics- and unsupervised learning-based smell detectors described below.

4.3.1 | Metrics-based detectors

A metric-based detector takes source code as the input, calculates a set of source code metrics that capture the characteristics of a given smell, and detects that smell by applying a suitable threshold [5]. In most cases, setting the threshold values is a highly empirical process, and it is guided by similar past experiences and hints from the metrics' author [32]. For example, as mentioned in Section 3, Sharma et al. [15] and Schwarz et al. [16] detect a similar smell called Insufficient Modularizatio if a configuration script contains more than 40 lines of code or an abstraction contains more than one class or define. Nevertheless, those thresholds do not hold to TOSCA because of the differences between these languages. Indeed, blueprints rarely contain a single type because of their nature, and types are usually small. In addition, no previous works on Blob Blueprint detection for TOSCA exist. Therefore, no hints are available from past experiences.

Statistical techniques can define a suitable threshold for each metric when no hints are available. In this case, we used the Interquartile Rule as in previous works [33] as a baseline:

$$T(x) = Q_3(x) + 1.5 \times IQR(x)$$

The formula defines threshold spotting blueprints representing upper outliers for a specific metric (i.e., smelly instances). It makes use of the third quartile (Q_3) and the interquartile range ($IQR(x) = Q_3(x) - Q_1(x)$) extracted from the blueprints selected for this tuning. Typically, a threshold is calculated for each metric, and a rule is defined that

[13] `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html`

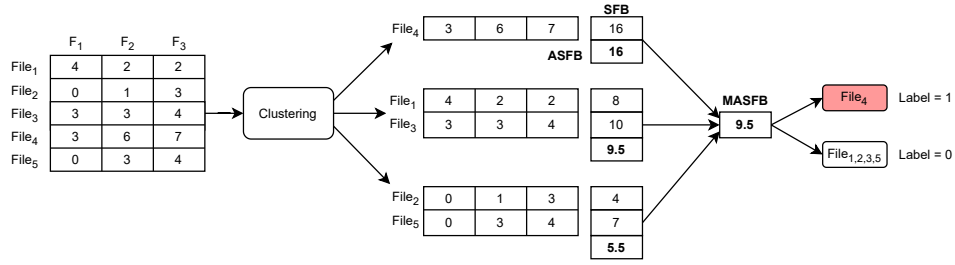

FIGURE 1 The cluster labeling scheme. Once clusters are defined, the feature values of blueprints in each cluster are summed (SFB). Then, the average SFB (ASFB) is calculated for each cluster. To support scenarios with multiple clusters (>2), the median values of these ASFBs (MASFB) are calculated so that clusters where $ASFB > MASFB$ are labeled as smelly and the remaining as sound.

combines them through logic operators. In this case we used the *logic* `OR`: an instance is detected as smelly if any metric exceeds the respective threshold. Conversely, the *logic* `AND` might be impractical, as the probability of detecting smelly instances drops when the number of metrics increases. For this reason, we relied on multivariate outliers to consider multiple metrics at once.

The standard method for multivariate outlier detection uses the Mahalanobis distance [34, 35], a measure of the distance between a point p and a distribution D. More specifically, it measures the number of standard deviations the point p is from the mean of D. The distances are typically interpreted by comparing the corresponding χ^2 value (with the degrees of freedom equal to the number of variables) to a cut-off p-value. Cases with p-value $< .001$ are likely to be considered outliers.

4.3.2 | Unsupervised learning-based detectors

The unsupervised learning detectors proposed in this study revolve around four popular clustering techniques available in the Python framework *scikit-learn* [36], namely *KMeans*, *AgglomerativeClustering*, *MeanShift*, and *BIRCH*. We relied on these techniques for their popularity[14] and because they are common in code smell detection for traditional source code [37, 38]. Furthermore, we used the implementations provided by *scikit-learn* to ensure easy operationalization for practitioners and replication for researchers. A detailed description of these techniques is available in the scikit-learn's official documentation.[15]

Number of Clusters. Most of the algorithms mentioned above require specifying the number of clusters in advance. Being that unavailable information, we resorted to the Silhouette coefficient [39] to validate the goodness of a clustering technique and select an appropriate number of clusters. It ranges between -1 and +1: a coefficient close to +1 indicates that the objects are well-matched to their cluster and poorly matched to neighboring clusters. A value close to -1 indicates too many or too few clusters, whereas a coefficient close to 0 indicates overlapping clusters. Therefore, we performed a randomized search on different hyper-parameters configurations for every clustering technique and retained the configuration that maximized the Silhouette coefficient across ten runs.

Cluster Labelling. The clusters resulting from the previous step have unique identifiers from a technical standpoint; these identifiers are not labels that indicate the cluster's smelliness. However, Zhang et al. [40] proposed a heuristic

[14] `https://dataaspirant.com/unsupervised-learning-algorithms/`

[15] https://scikit-learn.org/stable/modules/clustering.html

to label clusters in the context of defect prediction that Xu et al. [41] adapted to support scenarios with more than two clusters. Figure 1 depicts the heuristic that we instantiated for smell detection by changing the labels as follows:

1. **S**um up the **f**eature values of **b**lueprints in each cluster (SFB).
2. Calculate the average SFB for each cluster (ASFB).
3. Calculate the median of these ASFBs (MASFB).
4. Label every observation in each cluster as *smelly* if $ASFB > MASFB$; *sound* otherwise.

Step 4 assumes Blob Blueprints generally have larger values than sound blueprints for the considered metrics. Although this reasoning applies to defect prediction [40, 42, 43], we argue that it applies to the smell we investigated. Finally, step 4 labels the observations as sound in cases of one cluster.

4.4 | Performance Evaluation

We evaluated the techniques' performance on the ground truth in terms of *Precision* and *Recall* [44], defined as follow:

$$precision = \frac{TP}{TP+FP}$$

$$recall = \frac{TP}{TP+FN}$$

where TP is the number of smelly-instances classified as such by the model; TN denotes the number of non-smelly-instances correctly classified by the model; FP and FN measure the number of classes for which the model fails to identify the smelliness of classes by declaring these classes as smelly (FP) or non-smelly (FN).

Then, we computed the Matthews Correlation Coefficient (MCC) [45], a regression coefficient that combines all four quadrants of a confusion matrix, thus also considering true negatives. Its formula is:

$$MCC = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

where *TP*, *TN*, and *FP* represent the number of true positives, true negatives, and false positives, respectively, while *FN* is the number of false negatives. Its value ranges between -1 and +1. A coefficient equal to +1 indicates a perfect prediction; 0 suggests that the model is no better than a random one; and -1 indicates total disagreement between prediction and observation.

Please note that these measurements evaluate classifiers; however, they are applicable in this research since we apply binary clustering with known ground truth. In addition, we calculated the adjusted Rand index (ARI) – a standard metric for clustering evaluation. This metric measures the similarity between the ground truth assignments (e.g., true labels in our validation set) and our clustering approach assignments on the same samples (e.g., the labels assigned by our approach).

Finally, for each algorithm, we reported whether their MCC differs significantly. To this end, we applied the non-parametric Mann–Whitney U test [26] with a significance level $\alpha = 0.01$. To better control for the randomness of our observations, we used Bonferroni's correction [27] to adjust the significance level according to the number of comparisons (i.e., 15). Thus, the results are significant at the significance level $\alpha = 0.001$. P-values lower than this value show that the MCC of the two algorithms differs significantly.

Beyond the p-value interpretation, we calculated the effect size using Cliff's delta [28] to measure the magnitude of the difference between two populations and ranges from zero to one. For example, according to Kampenes et

TABLE 1 Overview of the features and results of the statistical analysis of the metrics. Mann–Whitney U test significant (*) if $p < 0.002$ (corresponding to a non-corrected $p < 0.01$ for each test).

Metric	Mean Blob	Mean others	U	Effect size
LinesCode	669	79	7381*	Large
NumTypesAndTemplates	15	3	6675*	Large
LCOM	12	2	6631*	Large
NumInterfaces	6	1	5303*	Medium
NumProperties	64	9	6638*	Large
NumImports	6	2	4699	Small

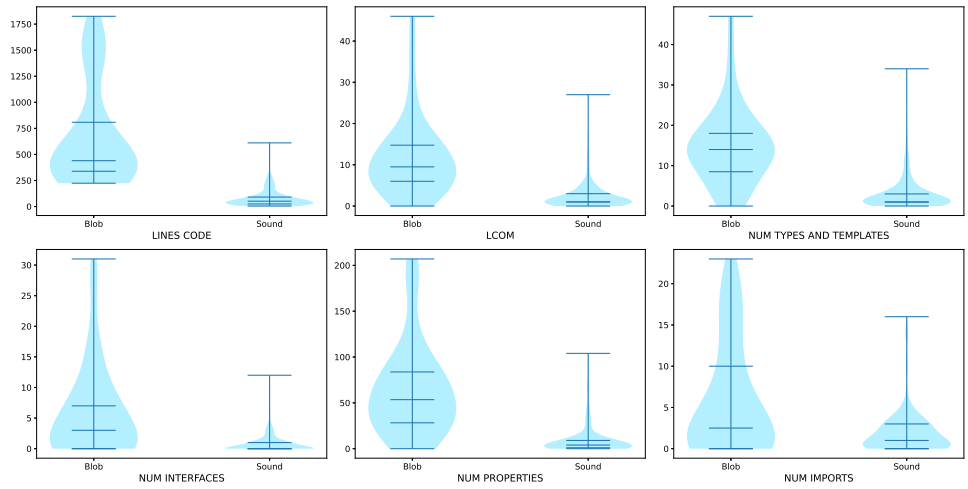

FIGURE 2 Results of the statistical analysis of metrics distribution: the violin plots show that the population of Blob and sound blueprints have different distributions for every considered metric compared with the ground truth.

al. [29], a value below 0.147 is considered trivial; between 0.147 and 0.33, it is small; between 0.33 and 0.474, it is medium, and it is large above 0.474.

Please note that we evaluated the techniques across 100 experiments to gain insights into performance variability. Each experiment considered a perturbed version of the original data set consisting of at least 290 uniformly sampled observations without replacement (i.e., a statistically relevant sample size) and reported statistics like median, mean, and standard deviation. We generated these versions upfront to evaluate all the techniques on the same data sets.

5 | RESULTS

Table 1 shows the metrics list and a statistical evaluation (as described in Section 4.2.1), while Figure 2 depicts an overview of their statistics and distribution through violin plots. As can be observed, Blob and sound blueprints populations have different distributions for all considered metrics compared with the ground truth. However, *NumImports* appears to be distributed independently from whether the blueprint is smelly or not, with a small effect size. Overall, this first analysis hints that their use, and the combination thereof, is a good proxy for detecting Blob Blueprints. Therefore, we used those metrics, except the number of imports, to build the smell detectors.

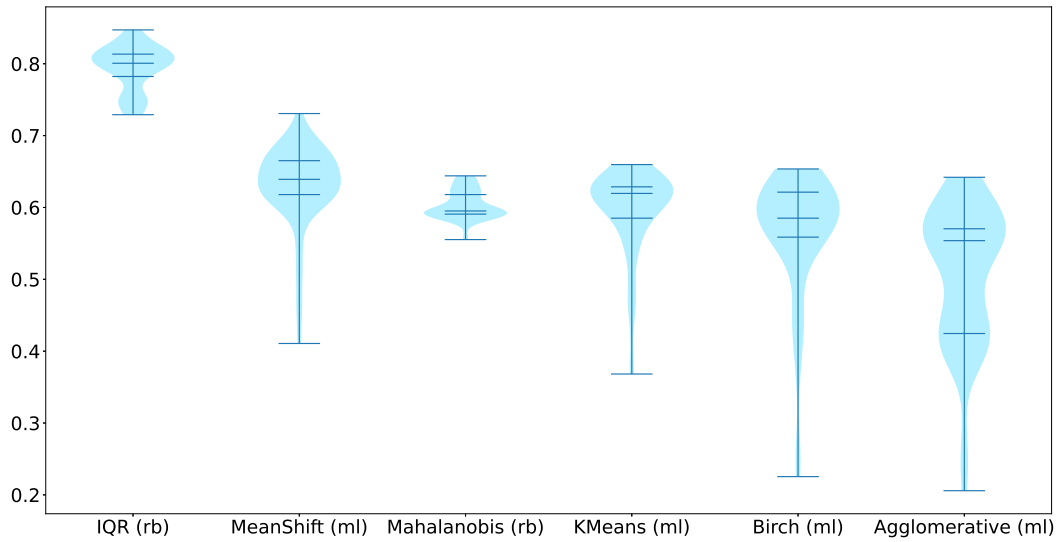

FIGURE 3 Matthews correlation coefficient across metric- and unsupervised learning-based detectors. The detector built using the *Interquartile rule* performs statistically better than those relying on unsupervised learning. Legend: (rb) = rule-based; (ml) = machine learning-based.

Summary of RQ$_1$: *All the considered structural code metrics, except NumImports, can distinguish between Blob and sound blueprints with medium to large effect size.*

Table 2 shows the results of our experiments for each detector in terms of MCC, Precision, Recall, F1, and ARI (for cluster-based detectors). Similarly, Figure 3 depicts violin plots to compare detectors' performance in terms of MCC. The average MCC ranges from 0.5 for the worst-performing detector, AgglomerativeClustering, to approximately 0.8 for the best-performing detector, i.e., the metric-based detector based on the interquartile rule. It is also the detector with the lowest standard deviation, alongside the other metric-based detector using the Mahalanobis distance. Thus, it yields the most stable results regardless of the metrics used; the minimum is below 0.73, the maximum is 0.85, and the standard deviation is 0.03. In addition, an unsupervised learning-based detector reaches the best precision, namely the one using the AgglomerativeClustering algorithm. However, the recall has a high drop-off. As can be observed by analyzing the F1, which summarizes precision and recall, the two metric-based detectors perform the best, with an average of 0.6 and 0.8. The best unsupervised learning-based detector, MeanShift, achieves slightly similar results. However, its performance is significantly worse than the best metric-based detector (22% worse in MCC), with a large effect size.

Table 3 shows that the differences among the detectors are, in most cases, very high and of practical significance. For example, all the detectors have 22% to 35% lower MCC than the best metric-based detector with a large effect size. However, most detectors generally reach moderate to high MCC and F1. These results are encouraging since false positives and false negatives are minimal. Please note that false positives and negatives might significantly impact users in the context of code smell detection. For example, false positives can have negative consequences, as developers put less trust in the tool when falsely alarmed multiple times for smells that do not exist. If this happens too often, developers could stop using the smell detector. False negatives, on the other hand, are comparably harmful. As the smell detector helps the developer during Quality Assurance, they might become less vigilant during code

TABLE 2 Average performance statistics across 100 experiments. Legend: (RB) = Rule-based; (ML) Machine Learning-based.

Algorithm	MCC		Precision		Recall		F1		ARI		FN	FP	TN	TP
	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std				
IQR (RB)	**0.79**	0.03	0.68	**0.05**	**0.99**	**0.02**	**0.80**	0.04	–	–	0	8	140	20
MeanShift (ML)	0.63	0.06	0.63	0.17	0.77	0.21	0.65	0.07	**0.52**	0.08	5	14	142	17
Mahalanobis (RB)	0.60	**0.02**	0.76	0.06	0.54	0.04	0.63	**0.02**	–	–	10	3	146	11
KMeans (ML)	0.59	0.06	0.59	0.10	0.72	0.12	0.64	0.07	0.51	**0.07**	6	11	143	16
BIRCH (ML)	0.56	0.09	0.64	0.14	0.63	0.20	0.59	0.13	0.48	0.11	9	8	148	13
Agglomerative (ML)	0.51	0.10	**0.82**	0.08	0.38	0.14	0.49	0.13	0.42	0.12	14	2	147	7

TABLE 3 Statistical comparison of mean MCC among detectors. The values below the diagonal are the differences between pairs of techniques in % (significant in bold). A negative value indicates that the detector in the row performed worse than the one in the column. The values above the diagonal are the effect size. Legend: (RB) = Rule-based; (ML) Machine Learning-based.

	IQR	MeanShif	Mahalanobis	KMeans	Birch	Agglomerative
IQR (RB)	–	Large	Large	Large	Large	Large
MeanShift (ML)	**-22**	–	Large	Medium	Large	Large
Mahalanobis (RB)	**-24**	**-5**	–	Small	Small	Large
KMeans (ML)	**-25**	**-6**	+2	–	Small	Large
Birch (ML)	**-29**	**-11**	**-7**	**-5**	–	Medium
Agglomerative (ML)	**-35**	**-19**	**-15**	**-14**	**-9**	–

reviews as they fully trust and rely upon the detector. The overall code quality can even decrease when it suffers from a high rate of false negatives.

> **Summary of RQ$_2$:** *The metric-based detectors perform statistically better than unsupervised learning-based detectors with large effect size.*

Finally, Figure 4 shows how the considered metrics impact the performance of each detector. During the stepwise forward selection described in Section 4.2.2, we tracked the variation in MCC, Precision, and Recall when adding features at each step. We added the metric that maximizes the MCC (or minimizes the drop in MCC) at each step. For example, the IQR-based detector (top-left graph) has the highest MCC when it uses only *LinesCode*. Adding *NumImports* decreases the MCC, although the combination *(LinesCode*, *NumImports)* maximizes the MCC among those consisting of two metrics, including *LinesCode*. Similarly, adding *LCOM* -– the best one among the remaining metrics — to them further decreases the MCC.

As observed, performance varies visibly across metrics sets and detectors. For example, the number of code lines maximizes MCC for the IQR-based detector. At the same time, performances are the lowest for the Mahalanobis-based detector. As for the unsupervised learning-based detectors, the same metric increases MCC only for MeanShift, while for the remaining, it decreases it. Precision tends to increase or stay constant. The opposite applies to recall.

In general, looking at the figure is clear that all the considered metrics improve performance. Therefore, one should consider these metrics together while building detectors. Please note that we could not analyze all combinations for time's sake. Nevertheless, we observed several metrics that recur across the optimal subset of features, shown in

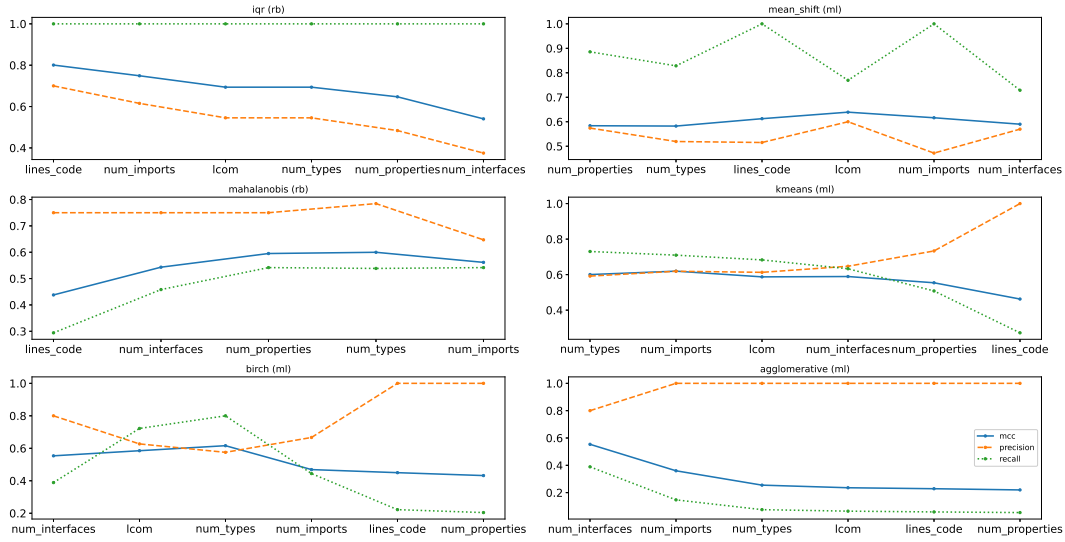

FIGURE 4 Stepwise forward selection to track the change in performance when adding the metric that maximized MCC (or, minimized its drop) at each step. For example, the IQR-based detector (top-left graph) has the highest MCC when only *LinesCode* is used. Adding *NumImports* decreases the MCC, although the combination (*LinesCode*, *NumImports*) maximizes the MCC among those consisting of two metrics and that include *LinesCode*. Similarly, adding *LCOM* – the best one among the remaining metrics – to them further decreases the MCC.

Table 4. Among them, the number of interfaces – analogous to the number of methods in traditional programming – is the one that occurs the most.

TABLE 4 Features that maximize MCC for each detector. Legend: (RB) = Rule-based; (ML) = Machine Learning-based.

Algorithm	Selected features
IQR (RB)	*LinesCode*
MeanShift (ML)	*NumTypes*, *LinesCode*, *LCOM*, *NumProperties*
Mahalanobis (RB)	*NumInterfaces*, *LinesCode*, *LCOM*, *NumProperties*
KMeans (ML)	*NumInterfaces*, *NumTypes*
Birch (ML)	*NumInterfaces*, *NumTypes*, *LCOM*
AgglomerativeClustering (ML)	*NumInterfaces*

Summary of RQ$_3$: *The number of interfaces appears to be a leading metric to maximize the overall performance. Follows the number of types and templates, code lines, and LCOM.*

6 | THREATS TO VALIDITY

This section describes the threats that can affect the validity of our study.

6.1 | Construct Validity

Threats to *construct validity* concern the relation between the theory behind the executed methodology and the found observations by assessing whether the observed outcome corresponds to the effect we think we are measuring.

In this work, we used source code measurements, which may not appropriately represent the intended characteristic by the researchers. We mitigated this threat by using two approaches. First, we looked for measures that have been empirically validated multiple times for traditional code smell detection and that could be ported to TOSCA. Then, we consulted the official TOSCA documentation to identify possible measures related to the blueprint complexity and size source. Second, we implemented these measurements following a test-driven development approach [46]: the developer first creates unit tests for the intended functionality. Afterward, they implement the measurement and improve it until all the initial unit tests pass.

Another threat relates to the construction of the ground truth, done manually. As manual work, such as labeling, can be prone to human errors, we acknowledge this as a possible threat to construct validity. We tried to mitigate this threat by summarizing the most prevalent definitions found in the literature for the analyzed smell. Furthermore, the authors of this study performed the validation, which poses a threat to the construction validity due to the bias regarding the perception of what metrics or quality attributes characterize the smell; involving external experts would mitigate this bias. However, it is worth noting that the authors involved in the validation have multiple years of experience in code quality and IaC research; although the annotators were not external, they could still be considered experts enough for this task. Finally, the annotators performed the validation *before* the subsequent analyses to mitigate additional bias; they also actively discussed their operations multiple times to reduce subjectivity.

Lastly, we collected GitHub repositories automatically based on a search string. Therefore, we may have missed relevant repositories due to a conservative search string.

6.2 | Internal Validity

Threats to *internal validity* concern the possibility that other factors could cause the outcome but were not measured during the research. A possible threat to internal validity is the source code measurement selection. It could be possible that other not included source code measurements could significantly influence the result. We mitigated this risk by selecting measurements that correlate with and can identify defective IaC scripts [47, 48].

6.3 | External Validity

Threats to *external validity* relate to the generalizability of the obtained results outside the scope of the research. We observed various threats to external validity in this work.

First, the obtained data set is considerably small. Although it includes a large subset of publicly available TOSCA blueprints, the size of the data set can still negatively affect the modeling performance of the clustering algorithm used.

Another threat might be that our dataset does not correctly represent the population of TOSCA blueprints because we could select only those publicly available. For example, the blueprints used in industrial contexts cannot be shared due to company regulations, but demo blueprints can. In that situation, our dataset could not represent the actual population of TOSCA blueprints.

Lastly, we only used four clustering algorithms. However, such algorithms are among the most popular and intuitive, easing operationalization and interpretability for practitioners. Although, less traditional algorithms might be

evaluated as well. For example, spectral clustering could be analyzed, given its ability to solve more complex situations, such as arbitrary non-linear shapes, as it does not make assumptions about the shapes of the clusters.

6.4 | Conclusion Validity

Threats to *conclusion validity* concern the appropriate usage of statistical tests and reliable measurement procedures, for example, to ensure the high quality of the conclusions. A possible threat to conclusion validity might be related to our work's implementation of the detectors used and the applied evaluation strategy. We followed the instructions given by previous authors [15, 16] for building metrics-based detectors. As for the unsupervised learning detectors, we used the implementation provided by the Python framework *scikit-learn* [36]. The metrics used to evaluate our clustering-based detection approach (i.e., Silhouette, Precision, Recall, and MCC) are widely used techniques for evaluating the performances of binary classification tasks.

Furthermore, the test used for the statistical analysis to estimate the difference between measures among detectors is a threat to the conclusion validity. There are many statistical tests whose choice relies upon the data structure, data distribution, and variable type, and the result can differ accordingly. To mitigate this threat, we applied a non-parametric test, which does not make assumptions about the data distribution, is commonly used in literature. Besides, since we conducted multiple hypothesis tests at once, there is a chance that at least one of the tests produced a false positive. Therefore, we used Bonferroni's correction to adjust the significance level to control the probability of committing a type I error and mitigate this threat.

7 | DISCUSSIONS, IMPLICATIONS, AND LESSONS LEARNED

In $\mathbf{RQ}_1$ and $\mathbf{RQ}_3$, we found that traditional source code metrics, such as the number of methods (*interfaces* in TOSCA), classes (*types* and *templates* in TOSCA), code lines, and lack of cohesion are good indicators of complex blueprints when mapped to their respective concepts in TOSCA. This result corroborates, on a technology-agnostic language, the findings of Sharma et al. [15] and Schwarz et al. [16].

Besides, $\mathbf{RQ}_2$ shows that practitioners should prefer metrics-based detectors to unsupervised learning-based detectors. The latter helps overcome shortcomings such as determining threshold values required by the former and possibly reduce the effort of collecting and identifying smelly blueprints. We believe that, despite the performance observed in this study, unsupervised-learning-based detectors can still play a role in detecting Blob Blueprints and other smells in TOSCA. However, a broader range of TOSCA blueprints and metrics may be needed to enhance them.

7.1 | Implications for Researchers and Practitioners

The results above pose several implications for researchers and practitioners, described below.

- **Implications for researchers:** There is still room for research in this area, and we argue for more empirical work on configuration smells to broaden our knowledge of complex blueprints and enhance the catalog of code smells for IaC. Our findings put a baseline to investigate which metrics should be used to detect Blob Blueprints. However, further research is needed to understand the relationship between the smelliness of the TOSCA code and the collected metrics. These results can lead to a better understanding of which features to utilize to improve code smell detection in TOSCA and enable the comparison of competing approaches.

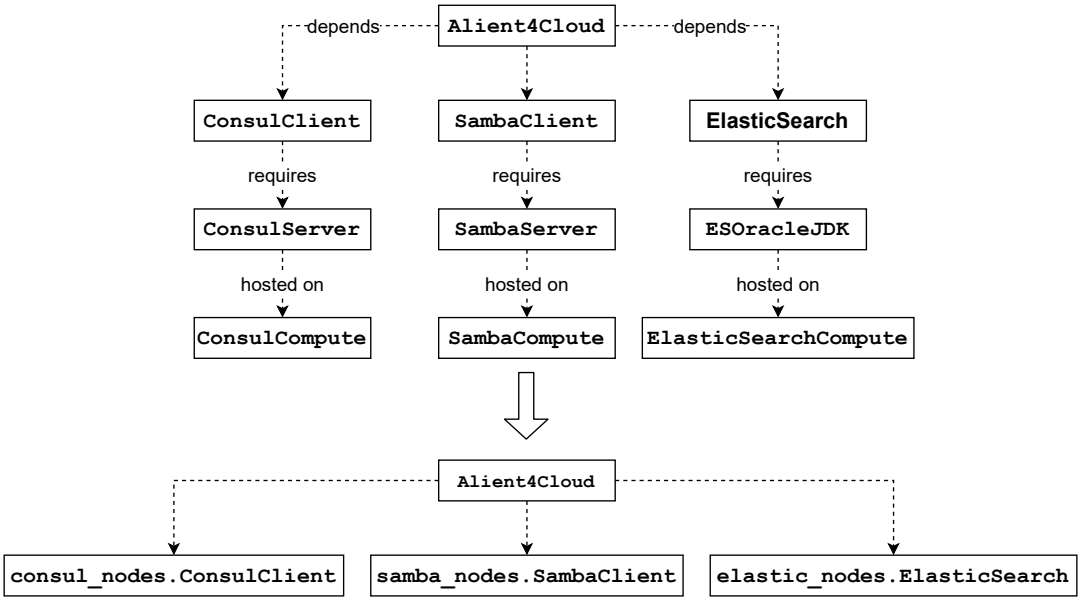

FIGURE 5 An excerpt of a Blob Blueprint with nodes targeting different technologies and a possible refactoring suggestion.

- **Implication for practitioners:** Practitioners can build upon our findings and shared material to implement novel methods and tools based on a small set of features such as those elicited in this paper. These tools will warn developers of complex blueprints and ultimately help reduce technical debt. For example, we already used those metrics as a proxy to predict failure-prone TOSCA blueprints within the scope of the European project called RADON, aimed at pursuing a broader adoption of serverless computing technologies within the European software industry.[16] One of the RADON key pillars is quality assurance for TOSCA. The results from analyzing blueprints in one of our partner's use cases within the project show that they help distinguish blueprints that may induce technical debt.[17] According to them, when keeping IaC quality, these metrics could ensure avoiding complex code representations.

7.2 | Lesson Learned

In addition to the implications above, we report some insights we observed while validating complex blueprints that we hope future researchers can benefit from to identify more fine-grained complexity measures for the *Blob Blueprint* smell.

- **Refactor nodes based on target technology.** During the validation, we observed several Blob Blueprints defining too many nodes, targeting different technologies. For example, a blueprint from *Alien4Cloud*[18] contains 16 nodes,

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

[17] https://radon-h2020.eu/wp-content/uploads/2021/09/D6.5-Final-Assessment-Report.pdf (Section 4.3.3.4)

[18] https://raw.githubusercontent.com/alien4cloud/csar-public-library/d08f5ac3f3f5279ad65fdf8c025459fafac37e75/org/alien4cloud/alien4cloud/topologies/a4c_ha/type.yml

a subset of which targets three different technologies: *Consul*, *Samba*, and *Elasticsearch*. Figure 5 shows those nodes and their dependencies. It might be advisable to refactor those types in different blueprints to reduce the complexity, each grouping type targeting the same or similar technology. Then, one should import those blueprints into the one at hand.

- **Refactor nodes based on types.** TOSCA provides types to describe the possible building blocks for constructing a service template. For example, node types to describe kinds of nodes, relationship types to describe possible relations among those nodes, and policy types to logically group TOSCA nodes that have an implied relationship and need to be orchestrated or managed together to achieve some result. While it is possible and might be advisable that a blueprint define one or more components of each type, we noticed that having too many different types makes the blueprint more challenging to understand. In this case, we suggest refactoring them in separate files for each type or group.
- **Move workflows into separate files.** Some of the analyzed blueprints had large workflows that contributed the most to increasing their size. Practitioners use workflows to automatically deploy, manage runtime, or undeploy TOSCA topologies. We suggest moving those workflows into different files and importing them into the current blueprint. Please note that although we noticed that large workflows could decrease the readability of a blueprint, we did not implement a measure for its size. The reason is that, among the collected blueprints, we observed workflows only in the project *Alien4Cloud*. In a preliminary investigation, we observed that this metric is too noisy, weighting the prediction of smelly blueprints towards those blueprints only.

8 | CONCLUSION

In this work, we enhanced the current knowledge of current practices in Infrastructure-as-Code and the detection of configuration smells. As indicated by Rahman et al. [7], current scientific works insufficiently address the characteristics of best practices within IaC, and only a handful of previous works investigated configuration smells. We conducted a study on the official OASIS standard for IaC called TOSCA, for which we constructed a comprehensive dataset of publicly available blueprints, deduced the characteristics of current practices, and investigated the performance of metric- and unsupervised learning-based techniques for smell detection. The implementation is made available on Github, accompanied by an explanation for usage and research reproduction.[19]

The main findings of this work are many-fold. First, we observed significant characteristical differences between smelly and sound blueprints based on their code structure for the current practices concerning TOSCA blueprint development. Our findings concerning configuration smells are also noteworthy. The range of researched configuration smells in previous work is relatively small because IaC is a new research area. However, we argue for more empirical work on configuration smells to broaden the smell catalog for IaC. Finally, other researchers can enhance this work based on the constructed dataset by applying more sophisticated techniques and analysis to investigate Blob Blueprints further and open opportunities for extensive studies on code smells in TOSCA.

Acknowledgment

Stefano, Gemma, Dario, and Damian are supported by the European Commission grant no. 825040 (RADON-H2020) and no. 825480 (SODALITE H2020). Fabio acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PZ00P2_186090 (TED).

[19] `https://github.com/jade-lab/tosca-smells`

References

[1] Ebert C, Gallardo G, Hernantes J, Serrano N. DevOps. *IEEE Software* 2016; 33(3): 94–100.

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

[3] Parnin C, Helms E, Atlee C, et al. The top 10 adages in continuous deployment. *IEEE Software* 2017; 34(3): 86–95.

[4] Fowler M. *Refactoring: improving the design of existing code*. Addison-Wesley Professional . 2018.

[5] Sharma T, Spinellis D. A survey on software smells. *Journal of Systems and Software* 2018.

[6] Guerriero M, Garriga M, Tamburri DA, Palomba F. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). ; 2019.

[7] Rahman A, Mahdavi-Hezaveh R, Williams L. A systematic mapping study of infrastructure as code research. *Information and Software Technology* 2019.

[8] Binz T, Breitenbücher U, Kopp O, Leymann F. TOSCA: portable automated deployment and management of cloud applications. In: Springer. 2014 (pp. 527–549).

[9] Lipton P, Palma D, Rutkowski M, Tamburri DA. Tosca solves big problems in the cloud and beyond!. *IEEE cloud computing* 2018.

[10] Baresi L, Quattrocchi G, Tamburri DA, Van Den Heuvel WJ. Automated Quality Assessment of Incident Tickets for Smart Service Continuity. In: Kafeza E, Benatallah B, Martinelli F, Hacid H, Bouguettaya A, Motahari H., eds. *Service-Oriented Computing*Springer International Publishing; 2020; Cham: 492–499.

[11] Li W, Shatnawi R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software* 2007; 80(7): 1120–1128.

[12] Palomba F, Bavota G, Penta MD, Fasano F, Oliveto R, Lucia AD. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 2018; 23(3): 1188–1221.

[13] Khomh F, Di Penta M, Guéhéneuc Y, Antoniol G. An exploratory study of the impact of antipatterns on software changeability. *École Polytechnique de Montréal, Tech. Rep. EPM-RT-2009-02* 2009.

[14] Sobrinho EVdP, De Lucia A, Maia MdA. A systematic literature review on bad smells — 5 W's: which, when, what, who, where. *IEEE Transactions on Software Engineering* 2018; 5589(c): 1–1.

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

[16] Schwarz J, Steffens A, Lichter H. Code Smells in Infrastructure as Code. In: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC). ; 2018: 220–228.

[17] Marinescu R. Measurement and quality in object-oriented design. In: 21st IEEE International Conference on Software Maintenance (ICSM'05). ; 2005.

[18] Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H. A bayesian approach for the detection of code and design smells. In: 2009 Ninth International Conference on Quality Software. ; 2009: 305–314.

[19] Fu W, Menzies T. Revisiting unsupervised learning for defect prediction. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering. ; 2017: 72–83.

[20] Li N, Shepperd M, Guo Y. A systematic review of unsupervised learning techniques for software defect prediction. *Information and Software Technology* 2020; 122: 106287.

[21] Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 1994; 20(6): 476–493.

[22] Ma Y, He K, Du D, Liu J, Yan Y. A complexity metrics set for large-scale object-oriented software systems. In: IEEE. ; 2006: 189–189.

[23] Honglei T, Wei S, Yanan Z. The research on software metrics and software complexity metrics. In: . 1. IEEE. ; 2009: 131–136.

[24] Basiri A, Behnam N, Rooij dR, et al. Chaos Engineering.. *IEEE Software* 2016; 33(3): 35-41.

[25] Cohen J. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 1960; 20(1): 37–46.

[26] Conover WJ. *Practical nonparametric statistics*. Volume 350. John Wiley & Sons . 1998.

[27] Weisstein EW. Bonferroni correction. *https://mathworld. wolfram. com/* 2004.

[28] Cliff N. Dominance statistics: Ordinal analyses to answer ordinal questions.. *Psychological bulletin* 1993; 114(3): 494.

[29] Kampenes VB, Dybå T, Hannay JE, Sjøberg DI. A systematic review of effect size in software engineering experiments. *Information and Software Technology* 2007; 49(11): 1073-1086.

[30] Mansfield ER, Helms BP. Detecting multicollinearity. *The American Statistician* 1982; 36(3a): 158–160.

[31] Catolino G, Palomba F, Fontana FA, De Lucia A, Zaidman A, Ferrucci F. Improving change prediction models with code smell-related information. *Empirical Software Engineering* 2020; 25(1): 49–95.

[32] Marinescu R. Detection strategies: Metrics-based rules for detecting design flaws. In: Proceedings of the 20th IEEE International Conference on Software Maintenance. ; 2004: 350–359.

[33] Aniche M, Bavota G, Treude C, Gerosa MA, Deursen vA. Code smells for model-view-controller architectures. *Empirical Software Engineering* 2018; 23(4): 2121–2157.

[34] McLachlan GJ. Mahalanobis distance. *Resonance* 1999; 4(6): 20–26.

[35] Filzmoser P, Reimann C, Garrett R. *A multivariate outlier detection method*. Citeseer . 2004.

[36] Pedregosa F, Varoquaux G, Gramfort A, et al. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research* 2011; 12: 2825–2830.

[37] Guggulothu T, Moiz SA. Detection of Shotgun Surgery and Message Chain Code Smells using Machine Learning Techniques. *International Journal of Rough Sets and Data Analysis (IJRSDA)* 2019; 6(2): 34–50.

[38] Azeem MI, Palomba F, Shi L, Wang Q. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* 2019; 108: 115–138.

[39] Rousseeuw PJ. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 1987; 20: 53–65.

[40] Zhang F, Zheng Q, Zou Y, Hassan AE. Cross-project defect prediction using a connectivity-based unsupervised classifier. *Proceedings - International Conference on Software Engineering* 2016; 14-22-May-.

[41] Xu Z, Li L, Yan M, et al. A comprehensive comparative study of clustering-based unsupervised defect prediction models. *Journal of Systems and Software* 2021; 172.

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

[43] Nam J, Kim S. Clami: Defect prediction on unlabeled datasets. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). ; 2015: 452–463.

[44] Baeza-Yates R, Ribeiro-Neto B, others . *Modern information retrieval*. 463. ACM press New York . 1999.

[45] Baldi P, Brunak S, Chauvin Y, Andersen CA, Nielsen H. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics* 2000; 16(5): 412–424.

[46] Beck K. *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam . 2002.

[47] Rahman A, Williams L. Characterizing defective configuration scripts used for continuous deployment. In: Proceedings of the IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). ; 2018: 34–45.

[48] Dalla Palma S, Di Nucci D, Palomba F, Tamburri DA. Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics. *IEEE Transactions on Software Engineering* 2021.