Detection of faults and code smells in (Cloud) Infrastructure as Code (IaC) templates

Kaarel Loide
Institute of Computer Science
University of Tartu
Tartu, Estonia
kaarel.loide@ut.ee

Abstract—Faults in the configuration of infrastructure can lead to serious consequences. The practice of Infrastructure as Code (IaC) aims to reduce this by introducing the concept of immutable infrastructure that is managed by a versioned source code repository. Since the configuration is stored as code, the question arises, whether errors could be further reduced by applying the known practices for code smell and fault detection from regular software development on IaC code bases as well. The goal of this paper is to help practitioners understand if IaC can be verified by commonly known practices from software development and what are the best tools currently available for keeping code in IaC projects secure and up to standard.

The author of the study analyses related work done by researchers to get an understanding of the state of the art in the academic field. In addition gray literature is analysed to see what practitioners commonly use.

Main findings of the paper are that some of the practices of code smells and fault detection are directly translatable to IaC, while others might need some tweaking. The academic research currently seems to be mostly interested in trying out machine learning approaches for both identifying new smells and best practices as well as detecting already known ones. As for the state of the industry there seem to be some tools that are gaining traction but there is still no consensual approach.

Index Terms—Infrastructure as Code; IaC; Code smells; Cloud computing; Fault detection;

I. INTRODUCTION

Finding ways to ensure high quality code in software development is a big and well known problem in the field. For Infrastructure as Code (IaC), this has been a largely unexplored area of research. Challenges arise from the fact that the field of IaC is still in the early stages of development. This is also the cause of a lack of agreed upon best practices and code smells. In addition to these challenges, the issue of language and tool diversity exists for IaC the same as it does for regular programming languages, making it even more difficult to develop standard practices [1].

A. Motivation

With the growing size of the software industry and the advancement of technology there has been an increase in the required speed of software development and delivery. While the pace grows so does the demand and expectations for the software developed. This makes the complexity and requirements set for new software systems a lot higher as they also need to be resilient and scale well under high load.

Human error is inevitable due to the increased development speed and growing application complexity. To help mitigate this, automatic error detection tools can be utilized. This is common practice for writing application code where automatic error detection is done in many forms. Application code is usually validated with different language specific linters as well as static analysis tools that identify code smells, bad practices and complexity issues.

In the case of IaC, there is a lack of clearly defined best practices and widely used tools for fault detection. The study at hand raises the following research questions to get a better understanding of this area.

• RQ1: What is the state-of-the-art in IaC template quality and fault detection?
  The aim of RQ1 is to find out what types of tools currently exist for the purpose of fault detection in IaC and from the results of the findings get an idea of what the major problems and blockers are in this area.

• RQ2: Can the approaches in code smells and programming faults be applied on IaC templates?
  Since the approach of code smell detection is well developed and widely used for regular application development RQ2 is raised to understand if the same principles could be applied for IaC code as well. If the same approach is valid for IaC then this could help develop the area in a more rapid pace as previous findings could be reutilised.

To find answers to the proposed questions the author performed a literature review by searching for relevant publications based on keywords. After the search, the author read through the titles and abstracts of the found publications and decided which ones to investigate further based on the relevance of the publication in regards to the research questions.

After the literature review additional information about tooling was gathered by searching for gray literature like blog posts, articles and documentation of tools. This was done to better understand the industry side of the matter.

The information gathered was then used to formulate answers to the research questions and to propose an approach that currently seems most suitable for industry use.

B. Structure

The remainder of this report is structured as follows. Section II gives a brief introduction into what is Infrastructure as
Code and what are code smells with some examples of the most popular ones. In Section III, the author describes the current state of the art from both the academic as well as the industry side. Different types of tools that currently exist for fault detection in IaC are analysed and the familiar approach of code smells from regular software development is discussed in the context of IaC. Next, Section IV proposes a solution and Section V answers the research questions. Finally Section VI summarises the analysis of the state of the art and concludes the paper.

II. BACKGROUND

The following section explains the main concepts, Infrastructure as Code and Code Smell, discussed in the paper in more detail.

A. Infrastructure as Code

Infrastructure as code is a way to consistently and automatically manage infrastructure spanning across a large number of both physical and virtual servers. It allows the codification of mission critical infrastructure operations. As all of the infrastructure operations are stored as code, version control systems can also be utilised to keep track of all the changes introduced into the system.

Another common practice in IaC is that the state of the infrastructure is not mutable by hand, meaning that the administrators are not able to modify anything by directly accessing the machines themselves. All of the state is controlled by the IaC tool used. Changes can only be introduced by modifying the code in the IaC repository and letting the tool handle creating the new state described in code. This is mainly done for two reasons. First reason is to have a reliable history of each change introduced. The second reason is to reduce the occurrence of a phenomenon called configuration drift. Configuration drift is when the state of the infrastructure becomes increasingly different from the configuration over time because of minor changes manually introduced into the system.

Most IaC tools are built upon some existing paradigms. For example Ansible is based on YAML and Ruby is used for Chef and Puppet. Some tools are also based on a custom domain specific language like Terraform, which uses the HashiCorp Configuration Language (HCL). An example of HCL can be seen on Figure 1. The use of these custom domain-specific languages slightly increases the difficulty of using already defined best practices for common languages in IaC.

Another common feature or IaC tools is the templating capability of the created scripts. This enables the developer of the script to inject custom variables into their code. The main purpose of this feature is to make the scripts reusable and provide a way to securely inject secrets into the code without saving them in version control repositories. For example Figure 2 shows a Jinja based template together with an Ansible script for using it to create configurations on servers.

Most IaC tools also provide some ways to invoke arbitrary scripts written in some other language like shell, powershell or Python from the tool itself, as seen on Figure 3, where a shell script is invoked from an Ansible playbook. By doing this, the flexibility of the IaC tool greatly increases but at the same time this flexibility introduced by the script invocation ability and templating also brings a drawback. The analysis of the IaC scripts becomes more complicated as the injected values by templates are unpredictable to static analysis tools and the arbitrary script executions also usually introduce a mix of different languages in to the equation. In addition to the different languages introduced script invocations also make prediction of the state achieved by the IaC script impossible in most cases, as the result of the script call is not easily predictable.
B. Code Smells

In regular software development there is a great effort of detecting faults as early on in the development cycle as possible, usually without ever even deploying the code into any server. This is mostly done through static analysis and different forms of testing.

In the context of this paper we will look into the possibility of applying fault detection through static analysis for IaC in a similar manner to regular software development. More specifically applying the idea of code smells and best practices to IaC.

The term code smell is used to describe some indication in code that is easy to spot but points to a deeper problem in a system. A smell is not inherently bad in and of itself, but the existence of a smell is often indicative of an underlying issue.

Code smells are a great tool to quickly provide feedback to the developer. As they are usually simple to spot by static analysis tools they provide feedback to the developer quickly but unlike the issues detected by syntax linters these smells usually need more manual intervention by the developer.

In software development code smells are most commonly divided into the following five larger categories [5].

1) Bloaters [6]: Code, methods and classes that have grown into such large proportions that they are hard to work with. These usually develop over a longer period of time when left untreated. Some examples would be: Long Method, Long Class and Primitive Obsession.

2) Object-Orientation Abusers [7]: Incomplete or incorrect use of object-oriented programming principles. For example: Switch Statements, Temporary Fields and Alternative Classes with Different Interfaces.

3) Change Preventers [8]: Patterns in code that make you want to change something in multiple places when trying to introduce a change in one place. Examples of Change Preventers would be: Divergent Change, Parallel Inheritance Hierarchies and Shotgun Surgery.

4) Dispensables [9]: Something unnecessary whose removal would make the code cleaner, more efficient and easier to understand. Some Dispensables are: Comments, Duplicate Code and Dead Code.

5) Couplers [10]: These are patterns that contribute to excessive coupling between different parts of code. Couplers are for example: Feature Envy, Middle Man and Message Chains.

By inspecting the descriptions of these smells it is obvious that some of them are easily applicable to IaC as well. For example, smells in the Dispensables category are something that will certainly translate into IaC templates because comments, code duplication and dead code can all occur in IaC. But in the case of most other smells, they are not so easily translatable and might need some adaption in most cases.

III. STATE OF THE ART

In the following section we will cover what is the current state of the art for fault detection in IaC by analysing related works and gray literature as well as proposing an approach that currently seems most ready to be used by practitioners in the industry.

A. Related Work

This subsection will describe the work previously done in this area. First, we will look into what types of best practices have been described in related literature and whether the approach of code smell detection is appropriate for IaC. And secondly we identify what types of state of the art tools have been developed so far for fault detection by the academic circles.

To get a better understanding of the best practices used Guerriero et al. [11] interviewed practitioners from the industry and their data revealed that 8-10 tools are about equal in their usage in the IaC space. This points to the fact that there is no best tool established yet. Users have said that each tool has their own strengths and weaknesses meaning that you just have to pick what is right for the problem you are trying to solve. Their research also discovered that although several best practices emerged from the interviews, they differ in the applicability based on the technology.

Even though the work of Guerriero et al. shows that code smells and best practices are still not that well developed, there also exists contradictory evidence. Multiple works point to the fact that the general approach of code smells can in fact be applied to IaC templates as well [1], [12]–[15]. The findings of Schwarz et al. indicate to the fact that IaC smells are in most cases technology agnostic and can in fact be defined independent of the underlying tool [1]. In conclusion it could be said that the answer to RQ2 is that the approaches in code smells and programming faults can to a certain degree also be applied to IaC templates.

As for the detection of faults and code smells, the maturity problem arises once again. The field is still in the early stages of development and there is a large number of different tools being used with no clear all around winner.

Two main problems seem to contribute to this lack of tools capable of detection faults in IaC. First of which is the diversity of languages used by the tools together with the fact that often multiple languages are interwoven in a single tool.

And secondly there is a lack of common best practices established as the area is still quite young. This makes the development of tools difficult as there is no common knowledge base for what issues to look for. There have been some
attempts at creating such tools but they have not become well
adopted or are still in the prototype phase.

For example Ting Dai et al. [12] from IBM created Secure-
Code in Python. They try to address the issue of arbitrary script
invocations in IaC by combining existing script analysers
like ShellCheck and PSScriptAnalyzer into a framework that
detects and prioritises faults found in scripts embedded into
Ansible files. They also have plans to extend SecureCode to
cover a larger number of IaC tools by adapting it for Terraform
as a next step. The created framework also provides a way to
plug in different analysers for the injected scripting languages
so if some IaC tool provides a way to inject Python code for
example, then this could be solved by plugging in a Python
analyzer to complement ShellCheck and PSScriptAnalyzer. As
a case study this tool was also introduced into the DevOps
pipelines at IBM to test 45 of their community service
repositories. Unfortunately the framework does not seem to
be available to the public and because of this, is not adaptable
by practitioners outside IBM.

Borovits et al. also created a tool for Ansible scripts
called DeepIaC. They attempted to use a deep learning based
approach to detect linguistic anti-patterns in Ansible scripts
and managed to achieve an accuracy of between 0.785 and
0.915. Currently DeepIaC helps to debug inconsistencies in
the names and bodies of Ansible code but there is also a plan
to extend DeepIaC to detect a wider range of bugs and work
with other IaC languages [13].

Another attempt has been made by Kumara et al. who have
taken a semantic approach to the problem of smell detection
in IaC. They created a prototype implementation for detecting
faults in TOSCA templates. The created tool uses GraphDB
as a knowledge base and SPARQL queries to identify smells
in the TOSCA files uploaded by the user. The result was
validated by three industrial case studies of the SODALITE
project. Existing TOSCA files were modified to contain smells
presented in the paper and validated with the defect predictor
to verify that each smell can be detected [14].

B. Solutions from the industry

In the following section we will cover some of the more
notable open-source tools mentioned in gray literature that
show potential to be useful for detecting faults and code smells
in IaC.

Checkov [16]: This is a Python based static code analysis
tool for IaC developed by BridgeCrew. Checkov scans files
for misconfigurations and security issues based on predefined
policies. The tool comes with more than 750 predefined
policies and also has support for creating custom ones using
either YAML or Python. The maintainers also accept custom
policies as contributions to the official policy base. An example
of the issues detected can be seen in Table I.

To achieve deeper analysis the tool internally creates a cloud
resource connection graph. This is used to find misconfigura-
tions across resource relationships.

Currently Checkov offers support for 7 different types of
IaC tools and frameworks:

- Terraform (for AWS, GCP, Azure and OCI)
- CloudFormation (including AWS OCI)
- Azure Resource Manager (ARM)
- Serverless framework
- Helm
- Kubernetes
- Docker

Open policy agent (OPA) [18]: A more general purpose
tool for enforcing policies across the entire software stack. OPA
is not specifically designed for detecting faults in IaC
but It can certainly be utilised for this use case.

OPA offers a high-level declarative language called Rego
that is used for defining policies. An example of a Rego policy
that denies the creation of Kubernetes Deployment objects
where containers are run as root can be seen on figure 4.
For enforcing the defined policies, OPA offers multiple ways
to integrate the tool into your existing stack. For example,
you could deploy OPA as an admission controller in your
Kubernetes cluster to deny the creation of misconfigured
resources [19]. OPA also offers a CLI to enable validating
configuration both in CI as well as locally on the machine of
the developer.

Conftest [21]: An extension of OPA is a tool developed
under the same project. Conftest allows the user to write
tests against structured configuration data and run these tests
using the Conftest CLI. The tool relies on the OPA Rego
language for defining policies. The main difference form OPA
itself is the fact that Conftest is an additional layer on top
of the OPA CLI. It offers a way to manage policies by
pulling them from different online repositories based on Git
or OCI, making sharing and managing policies easy across a
whole organisation. It also has many built in parsers for most
common languages used in IaC tools.

Fig. 4. Policy described in Rego language [20]
<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Entity</th>
<th>Policy</th>
<th>IaC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKV_K8S_2</td>
<td>resource</td>
<td>PodSecurityPolicy</td>
<td>Do not admit privileged containers</td>
<td>Kubernetes</td>
</tr>
<tr>
<td>CKV_K8S_8</td>
<td>resource</td>
<td>Deployment</td>
<td>Liveness Probe Should be Configured</td>
<td>Kubernetes</td>
</tr>
<tr>
<td>CKVDOCKER_1</td>
<td>dockerfile</td>
<td>EXPOSE</td>
<td>Ensure port 22 is not exposed</td>
<td>dockerfile</td>
</tr>
<tr>
<td>CKVDOCKER_7</td>
<td>dockerfile</td>
<td>FROM</td>
<td>Ensure the base image uses a non latest version tag</td>
<td>dockerfile</td>
</tr>
<tr>
<td>CKVAWS_18</td>
<td>resource</td>
<td>aws_s3_bucket</td>
<td>Ensure the S3 bucket has access logging enabled</td>
<td>Terraform</td>
</tr>
<tr>
<td>CKVAWS_24</td>
<td>resource</td>
<td>aws_security_group_rule</td>
<td>Ensure no security groups allow ingress from 0.0.0.0:0 to port 22</td>
<td>Terraform</td>
</tr>
</tbody>
</table>

File types currently supported by Conftest are:
- CUE
- Dockerfile
- EDN
- HCL and HCL2
- HOCON
- Ignore files (.gitignore and .dockerignore)
- INI
- JSON
- Jsonnet
- TOML
- VCL
- XML
- YAML

Both OPA and Conftest also have support for unit testing policies and allow for the creation of custom plugins that enable the user to extend the base functionality of the tool without modifying the source code.

IV. SOLUTION PROPOSAL

This subsection aims to answer RQ1 by considering both the academic research and state of the industry.

As best practices are in constant evolution and differ greatly based on the needs and requirements of the user, a highly configurable and extendable solution is needed.

Based on the research done. Academic efforts seem to currently still be too experimental for use in the industry. But based on gray literature currently the best all around approach seems to be using a mix of OPA and Conftest. This is because they share a common language for defining policies that could be used for multiple purposes. Some simpler fault detection could be done already in the developers machine by using the Conftest CLI or at least by the CI pipeline. While more complicated issues that need more context could be prevented by an OPA server integrated into the infrastructure of the application.

This 2-factor approach would reduce the friction of development by detecting mistakes earlier and providing faster feedback to developers while also not sacrificing the possibility to identify complex issues that are based on the current state of the application.

Another plus of using these tools is the fact that the policies created can be covered by unit tests. This makes the development process less error prone and increases long term maintainability.

To address the problem of different requirements in different contexts, both tools are extensible by the creation of custom plugins this provides developers the freedom to build their tooling and rules based on what is important to them.

Unfortunately neither of the tools provide policies out of the box. If there is no specific need for the flexibility of the Rego based tools then Checkov could be considered as an alternative as it already has a large number of predefined policies offered by the maintainers.

V. DISCUSSION

To conclude the state of the art we now summarise the previous discussion and answer the research questions based on the findings.

RQ1: What is the state-of-the-art in IaC template quality and fault detection?

By taking into account both the academic research and the state of the industry we can see that currently the state of the art is not completely agreed upon. Industry practitioners are mostly using simpler tools that are reliant on some sort of policy system to validate their IaC code. Most popular tools for the job seem to be OPA, Conftest and Checkov. This is somewhat different from the state of the art in academic research where the major focus seems to be on developing more experimental solutions that try to take advantage of machine learning or Semantic Web.

RQ2: Can the approaches in code smells and programming faults be applied to IaC templates?

For the most part it seems that the principles of code smells and programming faults can be applied to IaC as well. There are some contradicting points in related works but the majority seems to agree that the general approach is valid. By looking at the major categories of code smells we can also see that some of them certainly apply for IaC while others might need some adjustments to be suitable for IaC.

VI. CONCLUSION

Fault detection in IaC is still in the early stages of its life this means that there is still a lot to improve on in terms of tooling and best practices. Academic literature brings out conflicting points as to whether the principles of code smells can be applied to IaC. In general it seems as if at the very
least some practices can be directly translated while others need modifications to a certain extent.

As for the tools used. Researchers are experimenting with different novel approaches that are attempting to utilise machine learning or Semantic Web principles. These solutions are not ready for industry use though as they are still quite experimental. Based on gray literature the most prominent tools for practitioners currently seem to be Checkov, OPA and Conftest. These tools offer a variety of supporting functionalities in addition to the IaC validation, that are required by practitioners to be usable in the context of a larger organisation.

REFERENCES