Generalizing code analysis techniques to multiple IaC technologies

Nuno Filipe Marques Saavedra da Silva

Instituto Superior Técnico, Universidade de Lisboa
nuno.saavedra@tecnico.ulisboa.pt

Abstract. Infrastructure as Code (IaC) is the process of managing IT infrastructure via programmable configuration files. IaC has progressively gained more adoption in the DevOps landscape. Even so, IaC is not a silver bullet; akin to other software artifacts, IaC scripts can suffer from bugs. The scientific community has been active in proposing analysis methods which can mitigate the lifespan of such errors. However, the IaC technologies ecosystem is scattered and developing a tool for one technology will not solve the industry's problems for another. The main goal of this project is to create a technology-agnostic framework on top of which static analysis techniques can be developed. To this end, we propose an intermediate representation to which scripts from multiple technologies can be translated to. The developed analyses will then be applied to our intermediate model. We will focus our approach on configuration management of services tools, such as, Ansible and Puppet.

Keywords: Infrastructure as Code, Intermediate Representation, Static Analysis

Table of Contents

1 Introduction

Infrastructure as Code (IaC) is a tactic which has been progressively gaining more adoption in the DevOps world given its many advantages, such as, enabling scalable and reproducible environments or the traceability provided by version control software (e.g. git[1]). Guerriero et al. [8] define Infrastructure as Code as "*the DevOps tactic of managing and provisioning infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools*". The use of IaC scripts is essential to efficiently maintain servers and development environments. Rahman et al. state an example which illustrates the importance of IaC in large companies [29]:

> *Fortune 500 companies, such as Intercontinental Exchange (ICE), use IaC scripts to maintain their development environments. For example, ICE, which runs millions of financial transactions daily, maintains 75% of its 20000 servers using IaC scripts. The use of IaC scripts has helped ICE decrease the time needed to provision development environments from 1-2 days to 21 minutes.*

Although Infrastructure as Code has many benefits, by using it we are introducing a new class of possible issues — bugs in IaC scripts. As in software development, coding involves humans and humans are prone to make mistakes. Since one of the main problems in software engineering is the introduction of defects in code, a lot of effort has been made by the scientific community and companies in the industry to develop tools which try to avoid, identify or even automatically repair bugs. In recent years, efforts have been made in the same direction for IaC. For instance, Rahman et al. developed a tool, called SLIC, capable of identifying possible security issues in Puppet[2] scripts [29]. Sotiropoulos et al. developed a prototype which tries to identify missing dependencies between resources and services defined in a Puppet script [40]. Other examples include tools that find bugs and vulnerabilities in other IaC technologies, such as, Ansible[3], Chef[4], and CloudFormation[5] [13,18,30].

Existing tools that detect bugs in IaC code are very valuable. However, the great majority of work in this area has a limitation: even though some solutions may be easily extendable, they tend to be focused on a single technology. For instance, all the solutions mentioned above focus solely on a single IaC technology. In a recent study, which describes the state of the art related to DevOps research, Alnafessah et al. identified the need to surpass this limitation. In particular, they state that "*a research question is how to develop holistic and polyglot defect prediction and debugging environments for IaC*" [1].

The importance of surpassing the technology limitation comes from the diversity of tools used in IaC. By interviewing IaC experts, Guerriero et al. concluded that "*the IaC technology ecosystem is currently very scattered, heterogeneous and not fully understood, with no single tool dominating the market*" [8]. In

[1] `https://git-scm.com/` [2] `https://puppet.com/` [3] `https://www.ansible.com/`
[4] `https://www.chef.io/` [5] `https://aws.amazon.com/cloudformation/`

the same study, the authors identify the developers' need for IaC development tools, such as, IDEs, static analysis tools and security related tools. Considering these statements, it would be valuable to have an artifact capable of assembling multiple analysis techniques and generalizing them for different IaC technologies.

1.1 Work Objectives

Our main goal is to develop an artifact capable of assembling multiple analysis techniques and generalizing them to be applied in various IaC technologies. Although more technologies can be implemented in the future, we will focus our attention on technologies that deal with configuration management of services, such as, Ansible, Chef and Puppet. Our choice is related to the heterogeneity of tools for this purpose, and because the three tools mentioned are on the top of most adopted IaC tools (see Section 2.1.2). The artifact should work as a framework where researchers can develop new techniques, represent new IaC concepts or introduce a new technology. The represented concepts should work as an intermediate model to which techniques are applied. Scripts from supported IaC tools should be able to be translated to the intermediate model, allowing the framework to apply the same techniques to IaC scripts from different technologies. We will implement some techniques found in other studies (e.g. Sotiropoulos et al.'s technique for identifying missing dependencies [40]). We will answer the following research questions:

- **RQ1**: Is it possible to create a model which abstracts different IaC technologies? Are the concepts expressed in the model relevant enough to apply different analysis techniques from the literature to it?
- **RQ2**: What limitations do we find when creating a model which abstracts IaC concepts between different technologies?
- **RQ3**: Are we able to use the new framework to obtain similar results to the ones in already existing work?

It is important to mention that **RQ3** depends on a positive answer to **RQ1** since without a model, we are not able to develop the framework. However, we are positive that we will be able to develop a model which, even if not able to fulfill completely our expectations, will provide value to practitioners.

Framework requirements. The framework must be easily extendable in three different aspects: analysis techniques, the model to which those are applied, and the technologies supported. It must be carefully designed with software engineering principles that try to fulfill the extensibility goals. The intermediate model, on which the framework is based, should easily allow the introduction of new concepts and without disrupting other work already present in the tool. The model must also be able to capture similar concepts from different technologies, while assuring it is expressive enough to effectively apply analysis techniques from the literature to it. Capturing similar concepts may be difficult in some

cases since there are IaC technologies with very different purposes (e.g. Kubernetes[6] and Ansible). Even in tools with similar functionality, we may find issues. For instance, Weiss et al., while developing a tool called Tortoise to fix bugs in Puppet scripts, state that "*Tortoise leverages Puppet's DSL to model resources, which should be possible for languages like Salt, Ansible, and LCFG, but harder for Chef, a Ruby-embedded domain-specific language*" [41]. Another requirement we would like to fulfill is to allow pipelines of analysis techniques. When developing a new technique, it might be useful to use results from another existing analysis technique, add some more information from the model, analyze it, and reach a new conclusion. One possible example is to use a technique to identify the probability of a script being faulty (these techniques are described in Section 2.2.1) to set a threshold that defines if an error found by another analysis technique (a technique as the ones described in Section 2.2.2) in that script is considered or not. This approach could be used to try to decrease the number of false positives. We could also address performance issues by using the probability of a script being faulty to decide the set of analysis techniques applied to that specific script. For instance, more complex analyses would be applied to scripts with higher probability of being faulty. The pipeline approach follows principles of modularity present in software engineering: allows the reuse of information from existing techniques, and possibly simplifies complex analyses by breaking them into smaller pieces.

Novelty of our solution. To the best of our knowledge, no research has been done that allows to easily generalize analysis techniques to multiple IaC technologies. There are some tools, such as Tortoise [41] and SLIC [29] which could be extended to other technologies without much effort. Rahman et al. extended the study done with SLIC by creating a new tool, called SLAC, which also identifies security issues in IaC scripts. The new tool is capable of analyzing Ansible and Chef scripts instead of Puppet [30]. However, the authors were not able to reuse previous components from SLIC and had to develop a new "Parser" and "Rule Engine", which correspond to the two components that SLIC and SLAC have. Even inside the new tool, Ansible and Chef are handled separately. As SLAC, some other tools try to handle more than one IaC technology. Checkov[7] is a policy-based static analysis tool which is capable of analysing scripts from multiple IaC tools (e.g. Kubernetes and Terraform[8]). Although Checkov considers more than one technology and the model it uses has some abstracted concepts (e.g. attributes and a graph representing connections between nodes), policies must be created for each individual tool. We consider important to mention that these tools are only focused in a single analysis technique, and more flexible models would be necessary to extend them to more techniques. We can think of two main reasons why the technology barrier has not been addressed yet. First, IaC is a recent trending topic. By using data from Google Trends, Rahman et al. identified that "*interest in IaC has increased steadily after 2015*" [28]. Secondly, at first glance, the ratio between benefits and difficulties to solve

[6] `https://kubernetes.io/`
[7] `https://github.com/bridgecrewio/checkov`
[8] `https://www.terraform.io/`

the technology limitation may not look appealing to the scientific community. However, we consider the benefits to be extremely important to the industry, since the IaC technology ecosystem is very scattered, and research could become even more relevant if generalized for more than one tool.

1.2 Envisaged Contributions

The envisaged contributions of our work are:

1. A model capable of abstracting concepts between multiple IaC technologies.
2. A framework that enables the development of new analysis techniques for IaC and capable of generalizing them for a variety of tools.
3. An analysis tool with some techniques implemented and generalized for multiple IaC technologies.
4. A study about how well generalized approaches for IaC technologies may work.

We consider the framework to be our main envisaged contribution. Our objective is to create a standard to develop research about script analysis techniques in IaC. The framework will be free and open-source, and so the scientific community will be able to contribute to enrich the artifact. As the model gets richer, more complex and accurate techniques can be applied to it. As more technologies are supported, more value will be given to techniques developed in our framework. If successful, our work can have an high impact on future scientific research in this area.

1.3 Document Structure

Throughout this document we will explore the IaC ecosystem to understand how we can generalize analysis techiques to multiple IaC technologies. We begin in Section 2 by introducing the necessary background and the related work that supports our project. In Section 3, we will describe our solution to the problem of generalizing IaC analysis tools to multiple technologies, and in Section 4 we will describe how we will evaluate our solution and the timeline to complete the steps of our project. Finally, Section 5 concludes the document by summarizing our work.

2 Background & Related work

In this section we present the background and related work necessary to understand our solution and the motivation that lead us to try to solve the problem of generalizing IaC tools to multiple technologies. We start in Section 2.1 by giving context about what is Infrastructure as Code, its benefits, an overview of the tools ecosystem and a description of some of these tools. In Section 2.2, we will highlight the importance of support tools in software engineering and IaC, and we will describe analysis tools found on the literature to analyse IaC scripts.

Finally, in Section 2.3, we describe the importance of intermediate representations in a variety of Computer Science areas and how these representations are applied.

2.1 Infrastructure as Code

```
---
- name: Update web servers
  hosts: webservers
  remote_user: root

  tasks:
  - name: Ensure apache is at the latest version
    ansible.builtin.apt:
      name: apache2
      state: latest
  - name: Create default page file
    ansible.builtin.copy:
      dest: /var/www/html/index.html
      content: "Welcome, {{ ansible_default_ipv4.address }}!"
  - name: Start apache service
    ansible.builtin.service:
      name: apache2
      enabled: yes
      state: started
```

Fig. 1. Ansible Playbook which installs Apache, creates a HTML file to be provided, and enables the Apache service.

Infrastructure as Code (IaC) is the process of managing IT infrastructure using configuration files which can be written as normal code. In the modern world, where applications require high scalability and availability, IaC proves to be necessary by decreasing maintenance costs, reducing the risks of manual misconfiguration or inconsistencies by human error and allowing faster deployments and problem solving. The ability to version control the configuration of a system is one good example of the advantages in this approach since it enables all the beneficial practices already explored in software engineering. For instance, source code history allows implicit communication in a team and simplifies the process of pinpointing when and why a configuration error was introduced. One simple example to understand the power of IaC is the following: imagine that system administrators (sysadmins) have ten machines with SSH access and they want them to work as web servers. To achieve their goal, the sysadmins will need to install the Apache HTTP server[9], create a HTML file to show in the web page, and start the Apache service. Usually, they would connect to each

[9] `https://httpd.apache.org/`

machine, install the Apache package, create the file and start the service manually. However, this process may take some time, it is repetitive and it is prone to human error (e.g. inconsistencies between machines). Instead, it is possible to automate this procedure using a IaC tool such as Ansible. Ansible allows to configure machines by executing a list of tasks in each one of them. The configuration is done from a central point (another machine) which connects by SSH to each host and executes the commands to complete the defined tasks. The sysadmins could define the program in Figure 1 and execute it with Ansible to achieve their goal. Section 2.1.2 describes in more depth how Ansible works and its features.

2.1.1 Case studies

Case studies that highlight the benefits of IaC to companies have been addressed in the literature. Rahman et al. identified four studies which describe how IaC benefited information technology organizations [30]:

> *For example, the use of IaC scripts helped the National Aeronautics and Space Administration to reduce its multi-day patching process to 45 minutes. Using IaC scripts, application deployment time for the Borsa Istanbul, Turkey, stock exchange reduced from ∼10 days to an hour. With IaC scripts, Ambit Energy increased their deployment frequency by a factor of 1,200. The Enterprise Strategy Group surveyed practitioners and reported the use of IaC scripts to help IT organizations gain 210% in time savings and 97% in cost savings on average.*

A lot of other relevant examples can be found. Edgenuity is an organization which provides online learning and teacher resources. The company decided to change to the cloud in the beginning of 2020 and choose Chef to leverage this process. By using IaC scripts, what once would take the company's DevOps team 4 to 5 days to deploy, would now take less than 3 hours. The usage of Chef also allowed Edgenuity to scale from supporting 500,000 connections to 5 million when the COVID-19 pandemic hit, while minimizing disruption to the students [21]. The Germany's Federal Office for Agriculture and Food used Ansible to decrease time used in IT management and configuration by more than 50% [33]. Jewelers Mutual Insurance Company collaborated with Zivra to integrate Puppet in their workflow. They were able to "*reduce time to create an environment of about 30 servers from 4 to 6 weeks to under 1 day*" and increased consistency across their heterogeneous environments [24].

From the case studies above we consider there are two main takeaways. First, we can conclude that IaC technologies are widely used in the industry and with important benefits to the companies adopting them. Secondly, we can confirm the heterogeneity of the IaC ecosystem, since we have organizations choosing different technologies to the same purpose. Although Ansible, Chef and Puppet focus on the same class of problems, they have differences between them. These differences will be further explained in Section 2.1.2. Organizations, after analyzing the different options, end up choosing the tool which they consider more

fit to their goals. Edgenuity chose Chef "*because it was the only automated application packaging and delivery solution that was technology agnostic*" [21]. The Germany's Federal Office for Agriculture and Food state as reasons to use Ansible: "*firstly because of the quality of Red Hat's support, and secondly because the Red Hat Ansible Tower solution works best with our existing Red Hat environment*" [33]. Finally, Jewelers Mutual Insurance Company decided to work with Puppet "*because whether you're a developer or sysadmin, Puppet is intuitive and the learning curve is not steep*" [24]. There is not a standard technology to solve IaC problems, and so, it would be important to generalize support tools to multiple technologies.

2.1.2 IaC tools overview

Table 1. Categorization of IaC tools based on Guerriero et al.'s study [8].

Category	IaC Tools
Container Orchestration	Kubernetes, Nomad[a], Docker Swarm[b], Apache Mesos[c]
VM Management	Vagrant[d]
Configuration Management of Services	Ansible, Chef, Puppet, Saltstack[e]
Service Orchestration	Terraform, Pulumi[f], CloudFormation, Apache Brooklyn[g]
Image/Container Builder	Packer[h], Docker[i]

[a] `https://www.nomadproject.io/` [b] `https://docs.docker.com/engine/swarm/`
[c] `https://mesos.apache.org/` [d] `https://www.vagrantup.com/`
[e] `https://saltproject.io/` [f] `https://www.pulumi.com/`
[g] `https://brooklyn.apache.org/` [h] `https://www.packer.io/`
[i] `https://www.docker.com/`

The IaC ecosystem is composed by a high variety of tools without a current standard technology. Guerriero et al., in their study about adoption, support and challenges of IaC, state that the ecosystem "*is currently characterized by a plethora of different and often overlapping (in terms of their goals) tools and languages*" [8]. According to their study, no tool has more than 60% adoption and there are 10 tools with at least 20% adoption, indicating the lack of homogeneity when choosing IaC technologies. Adoption of tools with similar purpose can be understood by the trade-offs between technologies. These trade-offs can make a tool more appealing to some solutions than another tool with alike functionality. Ansible and Chef are examples of two tools with similar purpose; however, one main difference that we may find is the configuration language. Chef uses the Ruby DSL, which is "*aimed more at advanced developers rather than those with little to no programming experience*". Ansible uses YAML, which "*is relatively*

easy to learn, regardless of your prior experience". Given its simplicity, YAML is less powerful and, for that reason, it is not able to handle tasks as complex as Ruby DSL [4]. The trade-off between the learning curve and the complexity of tasks a technology can achieve is important when deciding which tool will be used for solving a certain problem. In the IaC ecosystem, there are also tools which fulfill different purposes. In the same study, Guerriero et al. mention that "*developers tend to select a stack of tools, each having a different purpose, whose combination enables full IaC development*". For instance, Terraform is usually used to provision infrastructure components, and Ansible deals with the configuration management of services. We may use Terraform to create some virtual machines and provide their configuration with Ansible (e.g. install an application in each one).

Considering the diversity of IaC tools with different purposes, it is important to categorize them. Table 1 summarizes the categorization done by Guerriero et al. in their study [8]. In addition to the examples provided by the authors, the table contains other tools which we consider to fit into these categories. The names of the added tools are underlined in the table.

In our work, we will focus our attention to configuration management of services tools. Two reasons lead us to select this class of technologies. First, the existence of scientific work to create analysis techniques for scripts in these tools allow us to test our framework by replicating these techniques and comparing the results to the original studies (more about this in Section 4) [29,30,40,41]. Secondly, tools that deal with configuration management of services appear to have the ecosystem with more heterogeneity and with more balanced adoption between technologies. In the study conducted by Guerriero et al., 4 tools in this category were mentioned to be adopted by the industry (Ansible, Chef, Puppet, Saltstack) [8]. Out of these 4 tools, 3 of them had at least more than 29% adoption by the IaC experts interviewed (Puppet - 29.5% / Chef - 36.3% / Ansible - 52.2%). Those 3 tools will be the ones we will further explore.

Ansible. Ansible is an open source infrastructure automation tool which uses a declarative language, called YAML, to automate IT tasks. As described in Ansible's documentation, the tool is able to "*configure systems, deploy software, and orchestrate more advanced IT tasks such as continuous deployments or zero downtime rolling updates*" [2].

Tool architecture. Ansible, unlike other configuration management tools, works with a push configuration setup. IaC tools usually have two types of machines: a server or set of servers where the configuration coded by sysadmins is maintained; and the nodes to be configured. In a push configuration setup, the sysadmin commands the server to provide the configuration to a set of nodes, instead of being the nodes asking the server for their configuration. Without direct user interaction with the nodes and with no initiative coming from them, push configuration technologies do not necessarily need an agent installed in each slave machine. That is the case of Ansible. A server (machine with access to all nodes), which maintains the intended configuration, accesses each machine,

usually through SSH, and pushes small programs, called Ansible modules, which execute the necessary actions to reach the desired state in the node. The list of nodes to be configured is maintained in an inventory. An inventory is usually a INI file where nodes are listed and can be grouped by being assigned a certain tag. An example of an inventory is shown in Figure 2.

```
[webservers]
www1.example.com http_port=80
www2.example.com http_port=303

[webservers:vars]
max_requests_per_child=808

[dbservers]
db0.example.com
db1.example.com
```

Fig. 2. Example of an Ansible Inventory with 2 tags (webservers and dbservers) and 4 hosts (ww1, ww2, db0 and db1).

Syntax and code structure. To define the desired state, sysadmins need to write code to achieve it. Ansible code is organized into three different hierarchical levels: playbooks, plays and tasks. A playbook is a file written in YAML which contains instructions to configure our nodes. A playbook is composed by a set of plays. Each play links a set of hosts to a set of tasks. Hosts can be identified by a tag to which they belong (selecting a group of nodes) or by their hostname. Tasks are the atomic building block of an Ansible configuration. A task has two parts: an optional field with the task name, which is useful because the name will be printed when the task is executed; the action the task itself will run, which is defined by selecting an Ansible module and providing the arguments that achieve the state we want. Ansible allows to group functionality into roles. Roles are folders which contain content such as tasks, variables, static files, modules or templates. The functionality of a role can then be reused in different playbooks by including the role in a play or task.

Variables/Attributes. "*Ansible uses variables to manage differences between systems*" [2]. Variables can be assigned a different value for each node, using the same name. These values can then be used in playbooks to change the behaviour of tasks according to the node they are being applied to. We can define variables in a variety of ways including creating them in playbooks or in the inventory. Figure 2 shows the definition of a variable in the inventory, called *http_port*, with a different value for each web server. In the same figure, we can also see the definition of a variable, called *max_requests_per_child*, that, instead of node specific, is defined for a group of nodes (*webservers*). Ansible has special variables related to information about the remote systems (nodes). These

variables are called facts and are automatically gathered by Ansible. Facts can give information, such as the IP address or the operating system of the node.

Execution order. By default, Ansible will execute the tasks by the order they are defined in the playbook. However, it is possible to define tasks that only run when notified. These tasks are called handlers and are triggered by a change in another task. For instance, we could define a task that changes a configuration file for a package, notifying a handler to restart the related service only if the content in the file actually changes.

Example and final regards. Figure 1 is an example of an Ansible Playbook with a single play containing 3 different tasks. In this case, the tasks use the apt, copy and service modules, which are builtin modules from Ansible. The playbook uses a fact to add the IP address of the node to the content of the HTML file. All the information presented is based on Ansible's documentation [2].

Chef. Chef has more than one product which helps IT automation: Chef Infra, Chef Habitat, Chef InSpec and Chef Automate. We will focus our attention on Chef Infra, since this tool is the one with direct correspondence to Ansible and Puppet. According to Chef's documentation, "*Chef Infra automates how infrastructure is configured, deployed, and managed across your network, no matter its size.*" [22].

Tool architecture. Like Ansible, Chef has servers, which store configurations, and nodes to be configured, in this case, called clients. A particularity of Chef is the existence of a workstation, which corresponds to the everyday computer the sysadmin works with. Chef Workstation is a set of tools, installed in the sysadmin's computer, that allow to test Chef's code and interact with other components of Chef Infra. Namely, *knife* is a command line tool capable of interacting with the server, which allows operations, such as, uploading configurations from a workstation. In contrast to Ansible, Chef Infra works with a pull configuration setup, requiring a client to be installed in each node. Periodically, each client contacts the server to retrieve the latest configuration for that particular machine, and, if it differs from the current one, the new instructions are applied to the node. Pull configuration setups have the advantage of automatically converging to the desired state when new nodes are added. If a workstation has SSH access to a node, we can add that node using the *knife* tool. This operation consists of installing the Chef client in the new machine and adding the node to the inventory maintained in the server.

Syntax and code structure. As any IaC tool, Chef also needs us to code the infrastructure configuration. The configurations are organized in "*fundamental units*" called cookbooks. Cookbooks are folders created with the *chef* command-line tool, which contain all the necessary content to define a certain scenario. We consider of particular interest recipes and attributes. Recipes are written using a programming language called Ruby and they define the steps necessary to configure part of a system. Recipes can have helper code, but their focus is

on the definition of resources. We can see resources as the equivalent of a task in Ansible. A resource is identified by his name and his type (package, service, file, etc...). The action and attributes associated with each resource define the desired state for the selected configuration item.

Variables/Attributes. Attributes are another component of a cookbook. An attribute is a key-value pair linked to a node. They can be defined in attribute files or in recipes. An example of an attribute that is linked by default to every node is the IP address. Attributes are relevant because we can use them to modify the definition of a resource accordingly to the node being configured.

Execution order. There are two things to consider when it comes to execution order in Chef: the order of resources and the order of recipes. Resources in each recipe are executed by the order they are defined in the file. However, similarly to Ansible, Chef allows a resource to notify another resource when its state changes. For that, we must identify the resource to be notified, the action to perform, and when to perform it. A resource can also subscribe to another resource, taking "*action if the state of the resource being listened to changes*" [22]. Recipes run in the order defined in a Run-list. A Run-list is an ordered list of recipes defined by the user to each node.

```
1  package 'apache2' do
2    action :install
3  end
4
5  file '/var/www/html/index.html' do
6    content "Welcome! I am " + node['ipaddress']
7    action :create
8  end
9
10 service 'apache2' do
11   action [ :enable, :start ]
12 end
```

Fig. 3. Chef Recipe which installs Apache, creates a HTML file to be provided, and enables the Apache service.

Example and final regards. Figure 3 is an example of a Chef recipe. In the example, we have 3 resources with 3 different types: package, file and service. We are configuring a web server which will provide the content in the *index.html* file. Each node configured by this recipe will show a different page since we are using the attribute *ipaddress* to set the content of the file. Considering, for instance, the language it uses to define policies, and the tools and possibilities it supports, Chef provides a more complex environment than Ansible. This fact may create difficulties when it comes to apply or generalize analysis tools to this technology. All the information presented is based on Chef's documentation [22].

Puppet. Puppet is the last configuration management tool we will explore. It has a lot of similarities with Chef, since the language it uses to program the configurations is also based on Ruby and the tool architecture is similar. The Puppet's documentation describes Puppet as "*a tool that helps you manage and automate the configuration of servers*" [23].

Tool architecture. Puppet has the same participants as the previous two tools. There are servers, usually called masters in Puppet context, and there are the nodes, called agents. Masters maintain the code and data to configure a node or a group of nodes. The data is managed by a component called Hiera, which will be better explained further ahead. As in Chef, Puppet also uses a pull configuration setup. Each node must have the Puppet Agent installed which communicates with the master, at regular intervals, to retrieve the latest configuration for that specific machine. Master and nodes communicate by HTTPS using SSL certificates managed by the master. Nodes are added to the deployment by submitting a certificate signing request (CSR) to the master, which has to be accepted by the admins. The agents use a library, called Facter, to collect information about themselves, information to which we call facts. When a node asks for its configuration, it also sends its facts, which will be used, together with the code and data the server has, to compile a catalog. A catalog describes "*the desired state of a specific agent node*" [23]. The agent receives its correspondent catalog and enforces the state described there. Finally, the node reports back to the server with information such as the events that occurred, metrics about the run, or the status of the resources in the node. Data such as reports, facts, catalogs and node information is stored in a component called PuppetDB. "*Storing data in PuppetDB allows Puppet to work faster and provides an API for other applications to access Puppet's collected data*" [23]. For instance, the data can be used by an analysis tool to assess vulnerabilities or other problems in the infrastructure.

Syntax and code structure. To write code in Puppet, we use Puppet's Domain Specific Language (DSL), which is based on Ruby. By using a DSL instead of Ruby itself, Puppet provides a more simplistic language and forces the users to a resource driven approach. Since the structure of the programs is well defined and much more limited, analysis tools tend to be easier to develop. In Puppet, we organize our code into modules. Modules, which can be seen as the same as cookbooks in Chef, are directories containing all the necessary content to manage a specific task in our infrastructure. Examples of content are configuration files, static files, tests, parameter defaults, or examples of how to use the module's functionality. Configuration files are called Manifests. Manifests can contain conditional logic, but their main goal is the definition of resources — the atomic building block of Puppet. The syntax for resources and their concept is really similar to Chef. A resource is declared by specifying a type, a title and a set of attributes, which describe the desired state for that aspect of the system. We can group resources into classes. Classes are named blocks which are able to configure larger chunks of functionality. Besides the benefit of being able to reuse groups of resources that are usually used together without having to write

them every time, classes can receive parameters to change the behaviour of those resources. It is also possible to create a hierarchy of classes, which is a useful concept from object-oriented programming.

```
class webserver ($content = "") {
  package { 'apache2':
    ensure => present,
  }

  file { '/var/www/html/index.html':
    ensure  => file,
    content => "${content} ${::facts['ipaddress']}",
  }

  service { 'apache2':
    ensure => running,
    enable => true,
  }
}

class{'webserver':
  content => "Hello from"
}
```

Fig. 4. Puppet Manifest which installs Apache, creates a HTML file to be provided, and enables the Apache service.

Variables/Attributes. Manifests can contain references to values that are specific to a node or group of nodes. We will call those references node attributes. When compiling a catalog, node attributes are replaced by the value correspondent to the node asking for its configuration. There are two types of node attributes: facts and Hiera key-value pairs. Facts, as explained above, are collected by the Facter and contain information such as the IP address or the operating system. Hiera allows to create hierarchies of attribute files. For instance, we could have attribute files specific for each node and attribute files for nodes with a certain operating system. Hiera could first search for an attribute in the node specific file and then, if the attribute does not have a value in there, in the correspondent operating system file. These files, written in YAML, contain the key-value pairs. Using Hiera allows us to detach data from code and gives us an easy way to change the behaviour of a manifest, accordingly to the node they are applied to.

Execution order. Puppet applies resources "*in the order they are defined in their manifest, but only if the resource has no implicit relationship with another resource*" [23]. In Puppet, we are able to specify relationships between resources and the tool is responsible for defining the execution order that fulfills those requirements.

Example and final regards. Figure 4 is an example of a Puppet manifest which configures a web server. The manifest has a class, called *webserver*, which contains the 3 resources necessary to configure it (install the Apache package, create the HTML file and start the Apache service). The class also has a parameter, called *content*, which defines the text that will be presented in the page. On line 17, we declare the class and set the content, enabling the resources in the class to be executed. On line 8, we use a fact to add the IP address of the node to the page. All the information presented is based on Puppet's documentation [23].

Differences between tools. Table 2 summarizes the differences between the 3 tools we described. It is important to mention the difference we consider between a procedural execution order and a declarative one. Procedural means that the code order is always respected and the developer is responsible for ordering the operations accordingly to what he expects. Declarative means that the developer only needs to define requirements for the execution order and the program is responsible to find the right execution to fulfill those requirements.

Table 2. Summary of differences between Ansible, Chef and Puppet.

IaC Tool	**Ansible**	**Chef**	**Puppet**
Configuration Setup	Push	Pull	Pull
Atomic Unit	Task	Resource	Resource
Add. agent	No	Yes	Yes
Syntax	YAML	Ruby	Puppet DSL
Execution Order	Code order/ Procedural	Code order/ Procedural	Code order/ Declarative

2.2 Support tools and code analysis

Tools that help the development of code and management of infrastructures are getting increasingly more important. As companies grow, there are more developers involved and code is produced/changed faster. Code bases also tend to get bigger, and so, more difficult to maintain. For instance, in January 2015, Google code base had "*approximately two billion lines of code in nine million unique source files*", and "*on a typical work day, (...) 16,000 changes to the codebase*" were made [20]. Managing this kind of scale requires a lot of human resources to code and make reviews which leads to high costs for the companies. With this rate of change, even with all the possible caution from developers, it is impossible to not introduce bugs into the code base. Some of these bugs may cause major faults. For instance, on June 2021, Fastly, a cloud-network provider with clients such as New York Times and The Guardian, "*experienced a global outage due to an undiscovered software bug (...) triggered by a valid customer configuration change*" [34]. Another example was the October 2021

Facebook Outage that caused products like Messenger, WhatsApp, and Instagram to become globally unavailable. The outage was caused by a command which was supposed to check the availability capacity of Facebook's global backbone network, but instead took down all its connections. Santosh Janardhan from Facebook stated that: "*our systems are designed to audit commands like these to prevent mistakes like this, but a bug in that audit tool prevented it from properly stopping the command*" [10]. The statement not only shows the need for tools that support these operations, but also the confidence developers and sysadmins have in these tools to help them.

A lot of effort has been made by the scientific community and companies in the industry to develop tools which try to support developers in their actions. For instance, IDEs (Integrated Development Environments), such as IntelliJ IDEA[10], help to avoid bugs by giving code suggestions, analysing and highlighting possible mistakes, supplying powerful refactor tools, and other supporting features. These features allow developers to focus on bugs related to functionality instead of worrying so much about simpler bugs, such as, typos or a variable name not changed in a refactor. Another example is Repairnator[11] which assembles multiple program repair approaches, coming from academic research, to create automatic and human-competitive patches to Java programs [15].

The increasing relevance of IaC and its parallelism with "normal" code lead researchers to also investigate how to develop techniques that identify or repair bugs in IaC artifacts. We divide research in this area into two groups: identifying faulty IaC scripts and detecting and repairing IaC errors. Inside these groups, there are multiple approaches explored.

2.2.1 Identifying faulty IaC scripts

When trying to identify faulty scripts, the objective is to give the users indications about where to spend their time improving code that might lead to problems in the future. Tools with this principle in mind should return a list of scripts, where each one is associated to a value representing the probability of the script being faulty. Techniques to identify faulty scripts usually rely on machine learning methods.

Rahman and Williams extracted characteristics of defective IaC scripts from qualitative analysis, which was applied to text features that appeared in faulty scripts. The text features were obtained by applying text mining techniques to convert Puppet scripts into tokens (words in the script). The characteristics associated to defective scripts that were found are: file-system operations, infrastructure provisioning, and managing user accounts. The authors applied to the tokens two techniques that generate new features capable of being used as input to predictive models: the BOW technique, and the TF-IDF technique. The Random Forest (RF) technique was used to build the models. The datasets to train them were created from crawling open-source projects, generating the features, and manually labeling defective scripts. 10-fold cross validation was used

[10] `https://www.jetbrains.com/idea/` [11] `https://github.com/eclipse/repairnator`

to evaluate the models. These models were able to obtain median F-Measure values between 0.70 and 0.74 depending on the dataset and text feature extraction technique [31].

Rahman and Williams, in another work, followed a similar approach but focused on source code properties like lines of code, number of attributes, or URL occurrences. They found that using these properties as input to predictive models outperformed the BOW technique from the previous work. The authors also tried to use other techniques to build the models such as Logistic Regression (LR) and Naive Bayes (NB). Depending on the dataset and evaluation metric, different techniques were better, but the Random Forest technique was outperformed in the great majority of the experiments. Rahman and Williams found that the properties with the strongest correlation to a script being defective are the number of lines of code and hard-coded strings. [32]

Palma et al. complemented the two previous works by addressing the problem that datasets with IaC scripts tend to have a number of defective samples much smaller than non-defective ones. To minimize the consequences of this issue, the authors used different techniques to build the models. These techniques use novelty detection, which means that the models are trained using only non-defective samples, and defective samples are identified as anomalies that differ from the data provided to the model (novelties). For the dataset used, the RF technique had a precision of 0.08 when these techniques had a precision up to 0.86, showing much better results. The authors also focused on Ansible instead of Puppet but stated that they planed to "*investigate how novelty detection generalizes on software defect datasets from different configuration management languages*" [19]. The dataset used was built by the *RADON framework* created by Palma et al. [18].

Palma et al. developed a framework called *RADON Framework for IaC Defect Prediction* which is "a *fully integrated Machine-Learning-based framework that allows for repository crawling, metrics collection, model building, and evaluation*". The study conducted with this framework concluded that, when considering individual projects/repositories as the training base for a model, the Random Forest technique outperformed techniques such as Support Vector Machine and Logistic Regression, "*regardless of the metrics used or the project's characteristics*". The authors also concluded that IaC-oriented metrics (lines of code or number of attributes) achieve better results than process metrics (total number of lines added or number of developers that changed a file) or delta metrics (amount of change in a file). The *RADON Framework* is capable of categorizing IaC scripts as possibly faulty or not, without manual intervention, by analyzing commits and issues in the project's repository. Palma et al. show their interest to extend their approach to other IaC tools and languages [18].

2.2.2 Detecting and repairing IaC errors

Approaches that try to detect and repair IaC errors have the goal of minimizing faults caused by human error. These tools can also give more confidence to the artifacts being deployed, increase awareness of the users about what they

should not do, or analyze a large quantity of scripts when the scale does not allow humans to do it. These approaches are more diverse than the ones to identify faulty IaC scripts.

Rahman et al. applied qualitative analysis on Puppet scripts to identify seven security smells, such as, *hard-coded secrets* and *use of HTTP without TLS*. The qualitative analysis and the smells identified allowed to create the rules for a tool called SLIC, which detects security smells in Puppet scripts. The authors divided SLIC into two components: a parser and a rule engine. The parser goes through a IaC script and returns a model (set of tokens) to which the rule engine applies its rules. The rules are based on the identification of string patterns in the tokens generated by the parser. These string patterns are extracted from the manual qualitative analysis. The authors submitted 1000 randomly selected occurrences of security smells identified by SLIC and out of the 104 responses they got, 64.4% of the practitioners agreed with the issues identified [29]. Rahman et al. replicated this work to Ansible and Chef where they founded two more security smells: *missing default in case statement* and *no integrity check*. Although we can think of reusing the rule engine and only changing the parsers when adapting the previous work to new languages, the authors considered that they had to redo both components [30].

Sharma et al. used the knowledge of traditional software engineering and best practices associated with code quality management to leverage the creation of a configuration smells catalog for Puppet scripts. The catalog is composed of 13 implementation (e.g. improper alignment of arrows or long statements) and 11 design configuration smells (e.g. insufficient modularization or duplicate block). The authors used Puppet-Lint[12] to detect the majority of implementation configuration smells and developed a tool, called Puppeteer, to detect the design smells. Using these tools, the authors analyzed 4621 Puppet repositories with 8.9 million lines of code to answer questions such as the distribution of maintainability smells in configuration code. They found *Improper Alignment* to be the most detected implementation smell and *Insufficient Modularization* the most detected design smell [38]. Shcwarz et al. extended the research done by Sharma et al. by applying the detection of IaC smells to another technology, namely Chef. The authors also categorized IaC smells as technology agnostic, technology dependent or technology specific, and introduced new code smells from the field of software engineering, such as, *Long Resource* and *Too many Attributes*. Finally, by applying the code smells to repositories with Chef scripts, the results allowed to conclude that "*these smells are adequate to be used to investigate the quality of IaC in general*" [36].

As in the work to identify security smells [29], Rahman et al. also applied qualitative analysis, but instead of code snippets, the authors used defect-related commits with the goal of identifying defect categories for IaC scripts. They identified 8 categories with *configuration data-related defects* being the most frequent category, and *idempotency* being the least frequent one. The authors used the information they got from the qualitative analysis to create empirical rules that

[12] `http://puppet-lint.com/`

automatically identify defect categories in enhanced commit messages (ECMs). ECMs are the combination of commit messages with bug report descriptions that are linked to the commits (e.g. issues). These rules were used to build a tool, called ACID, with a average precision and recall of 0.84 and 0.96, respectively, across all categories [27].

Chen et al. took a different approach to identify error patterns. In their paper, the authors extract error-fix-induced code changes from historical commits, and then used a clustering algorithm, namely HDBSCAN, to group similar code changes. By manually analyzing the clusters, they identified error patterns in the scripts, which were grouped into categories, such as, operating system related errors and file related errors. The authors then used the identified error patterns to propose a set of rules which were implemented in a tool called *Puppet Analyzer* [5].

Borovits et al. created an approach, called DeepIaC, to identify inconsistencies between task names and task bodies of Ansible scripts by leveraging word embedding and deep learning. The models, created using Convolutional Neural Networks (CNNs), classify a task as consistent or inconsistent and were able to obtain an accuracy between between 0.785 and 0.915 [3].

A problem that may emerge in IaC scripts is the lack of dependencies between resources. Shambaugh et al. addressed this problem by creating a tool, called Rehearsal, that checks if Puppet manifests are deterministic and idempotent. The authors translate IaC scripts into an intermediate language that models resources as the description of their file-system transformations. Logical formulas are created based on the semantics of each language's primitive and of the manifest's resource graph (representation of the valid sequences to apply the resources). By providing these formulas to a SMT solver, the solver decides if the program is deterministic or not. However, since the analysis is static, some transformations can not be identified, namely in manifests where the *exec* resource is used to run embedded shell scripts [37]. Sotiropoulos et al. take a different approach to the dependencies problem which allows to solve the shell scripts issue. Their method "*collects the system calls invoked by a Puppet program*" and uses them to model the file-system transformations in an intermediate representation, called *FStrace*. The Puppet resources are linked to the correspondent system calls, which allows to infer the relationships between resources by interpreting the semantics of the FStrace program. These relationships are compared to the ones in the dependency graph, which is defined in the Puppet manifest, to check if there are dependencies missing [40]. Their work also identifies missing notifiers, which were not covered in Shambaugh et al.'s work [37].

Weiss et al. created a tool, called Tortoise, that aims to avoid configuration drifts in Puppet scripts when sysadmins use the shell to fix configuration errors. Tortoise records system calls and file system changes caused by shell commands. After that, the tool uses the recorded information and the original Puppet manifest to build a model in a language "*with primitive operations that manipulate files*". The model is translated to logical formulas that are solved by a SMT solver, generating possible patches to Puppet scripts. Constraints coming from

the manifest are considered as soft constraints and the ones coming from shell updates are seen as hard constraints. This allows to update the manifests while trying to minimize changes. The patches are then ranked in a way that favors repairs with fewer changes. By doing experiments with 42 scenarios where the manifests would need repair, the authors identified that the "*the highest-rank repair Tortoise synthesized was the correct repair 76% of the time, and the correct repair was in the top five Tortoise-synthesized repairs 100% of the time*" [41].

Ikeshita et al. used a mixed approach that uses both test suites and static verification to check if a Chef program is idempotent. The approach uses an intermediate model that describes file-system manipulations. The model is used to do a static verification capable of reducing the number of necessary tests to check the idempotence of the program [9].

2.3 The role of intermediate representations

Intermediate representations allow the creation of abstractions which are useful for a variety of purposes. For instance, an intermediate representation may allow to abstract concepts to make the job of reasoning about an analysis easier. Although many more purposes may be found, considering the goal of our work, we will focus in the abstraction of similar concepts from multiple representations into a single intermediate representation. In this case, if we have an analysis capable of being applied to the intermediate representation, and translators from the original representations to the intermediate one, we can generalize the analyses to all the original representations. Many areas of research in Computer Science benefit from the usage of intermediate representations, we will describe some of them below.

Silva et al. proposed a tool, called RefDiff 2.0, to identify refactoring operations in source code with a language-agnostic design. The tool is capable of identifying refactors in Java, Javascript, and C and more languages can be supported by adding plugins to the system. The authors' approach was to create an intermediate representation, called Code Structure Tree (CST), that abstracts the specificities of each programming language. A CST is a tree-like structure which is focused on coarse-grained code elements, such as, classes or functions, and in the relationships between them. The nodes of the tree contain the following information about a code region: identifier, namespace, parameters list, tokenized source code, tokenized source code of the body and node type. The node types vary with the programming language's characteristics. For example, in C, the authors defined file and function as the only node types, while in Javascript the class type also exists. Although different types between languages exist, analysis rules do not consider specific types but only relations of equality or inequality between node types, following the language-agnostic approach. When it comes to relationships, the CST can represent calls (e.g. a method calling another method) or a hierarchy (e.g. a class contains a method or a class extends another class) between nodes. All the analyses are then executed using the CST,

which allows to work with different programming languages by only implementing the translator to this representation. Using a dataset of real refactorings in Java, RefDiff 2.0 got a precision of 0.96 and a recall of 0.80, which although not better than the state-of-art, it achieved similar enough results considering the language-neutral approach. The evaluation for Javascript and C obtained a precision of 0.91 and 0.88 and recall of 0.88 and 0.91, respectively for each language [39].

With the advance of software defined networking (SDN) in network management, many network programming languages (NPLs) have been proposed that allow operators to very efficiently program network data planes (NDPs). A NDP is the layer of a network device responsible for the forwarding of packages. In SDNs, the NDPs functionality is delivered through software. Considering the variety of NPLs and NDPs, Li et al. proposed an intermediate representation to express NPLs, called Network Transaction Automaton (NTA). The creation of this representation addresses two problems mentioned by the authors: the lack of interoperability between programs written in different languages, and the dependent evolution of NPLs bound to specific NDPs. To achieve compatibility between each NPL and all the NDPs and vice-versa, an architecture was proposed with NTA as its core. NPLs have a translator (*frontend*) to the intermediate language, and NDPs have a compiler (*backend*) from NTA to their correspondent representation. This architecture allows NPLs and NDPs to evolve independently. An NTA is an automaton with edges associated to network transactions. Network transactions contain the necessary information to model the classes of NPL semantics considered by the authors. These transactions can be represented as three-tuples with elements representing the next hop to be forwarded, the consumption of network resources, and a stateful operation capable of checking and updating node variables. Programs are translated to NTAs which are then composed using customized automaton operations. The composition of programs, which can be written in different languages, enables interoperability between NPLs. Li et al. showed that NTA can express the semantics of 6 NPLs used in the industry and is capable of efficiently compose the translated programs without semantic loss [14].

Another field of Computer Science where intermediate representations are present is in the formal verification of programs. Programs written in regular programming languages, like Java and C, are translated to intermediate verification languages (IVLs), such as Boogie [11] and Why3 [6]. The intermediate language works as a separation between the backend and frontend of program verifiers. Leino described both components of this architecture: "*front end is concerned with breaking down the semantics of given source-language programs into the more primitive operations of the intermediate language, and the back end is concerned with encoding the meaning of the intermediate program as efficient theorem-prover input*" [12].

As described in Section 2.2.2, some analysis tools for IaC also use intermediate representations [9,37,40]. Namely, these tools use intermediate models to describe file-system manipulations done by IaC scripts. These manipulations in-

clude creation and removal of files or directories, creation of links or renaming a file. Shambaugh et al. translated IaC scripts to the intermediate representation by mapping types of resources to their filesystem operations [37]. Sotiropoulos et al. used system calls executed by each resource in a IaC script to automatically map the resources to the filesystem operations, which were represented in the intermediate language [40]. Even though the main objective of the authors in these works was not the translation of scripts of multiple IaC technologies to these intermediate languages, it is possible and would allow the execution of the same types of analysis to other technologies.

3 Solution

The main goal of our work is to develop a framework where multiple analysis techniques can be developed and easily generalized to be applied in various IaC technologies. The framework will also work as the tool to run these analyses on IaC scripts.

3.1 Architecture overview

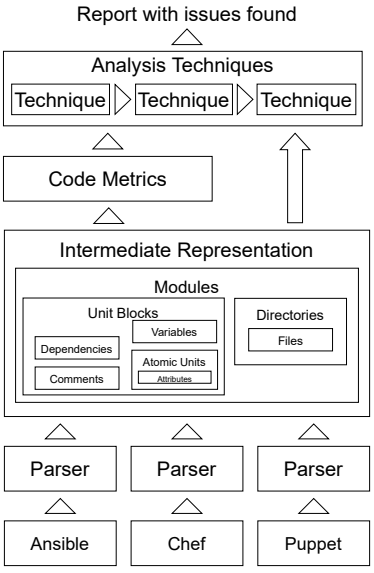

Fig. 5. A simplified architecture of the framework we will develop.

	Ansible	**Chef**	**Puppet**
Modules	Roles	Cookbooks	Modules
Unit Blocks	Playbook	Recipe	Class
Atomic Units	Task	Resource	Resource

Table 3. The table describes what component of each IaC tool corresponds to a component of the intermediate representation.

Figure 5 describes the architecture that we propose to build our framework. We consider it a simplified architecture since the intermediate representation

may change accordingly to the needs we will find while developing the framework, and details of implementation are not considered. The architecture considers 6 components: **(1)** the scripts of the supported IaC technologies which will be used as input to technology-specific parsers (see Section 3.2); **(2)** parsers for each technology which will transform IaC scripts into an intermediate representation; **(3)** the intermediate representation to which the analyses will be applied (see Section 3.4); **(4)** code metrics used by the analysis techniques and that are calculated using the intermediate representation (see Section 3.5); **(5)** analysis techniques which by being applied to code metrics and the intermediate representation will identify issues in the scripts (see Section 3.3); **(6)** a report that summarizes the issues found. In the following sections, we will explain the decisions we made and go deeper into the way each component works.

3.2 Supported IaC technologies

As described in Section 1.1, we will focus our attention to configuration management of services tools. Namely, we will start by implementing our approach to Ansible and Chef. We chose these two tools because when considering the three most popular configuration of services tools, Ansible, Chef and Puppet (described in depth in Section 2.1.2), Ansible and Chef are the ones with more differences. For instance, Ansible has a task-driven approach and the language used has a simple syntax, while Chef has a resource-driven approach and uses Ruby, a powerful and more complex programming language. If we can efficiently generalize analysis techniques for these two tools, we can show the value of our approach. Later, if the time constraints allow us, we will also implement a parser to Puppet manifests. Section 1.1 mentions the reasons to the focus on configuration management of services tools which are further corroborated in section 2.1.2. These reasons are related to the heterogeneity of the configuration management tools ecosystem and the high level of adoption these tools have in the industry.

3.3 Analysis techniques

Since the components of the intermediate representation we will create are driven by the needs of the analysis techniques we want to support, we start by enumerating these analyses. We chose from the set of analysis described in Sections 2.2.1 and 2.2.2 the analysis techniques with higher relevance and that use concepts we consider generalizable. We considered only works that are the state-of-art in the analysis of a certain type of issues. We will not consider dynamic analysis techniques, such as the ones presented in the works of Sotiropoulos et al. [40] and Weiss et al. [41], because we want to base our intermediate model in information contained in IaC scripts (static analysis) instead of other sources. Even though we will not consider these techniques, our framework should be able to support the objectives of analyses presented in these works (e.g. find missing dependencies), if in our architecture we are able to create a static analysis to fulfill the same goals. We think that dynamic approaches would be harder to

generalize since different analysis techniques would require different information. Besides that, more work would be necessary for each supported technology. For instance, if we wanted to apply Sotiropoulos et al.'s approach, we would need to be able to run IaC scripts from different technologies and map the system calls to IaC components of those technologies. Finally, we selected the following works:

1. "*The Seven Sins: Security Smells in Infrastructure as Code Scripts*" by Rahman et al. [29]
2. "*Code Smells in Infrastructure as Code*" by Schwarz et al. [36]
3. "*Singling the Odd Ones Out: A Novelty Detection Approach to Find Defects in Infrastructure-as-Code*" by Palma et al. [19]

The papers are enumerated by order of implementation priority. We will start by implementing analysis techniques with more proofs to be generalizable. By doing this, we are able to check the value of our approach as the uncertainty increases. Rahman et al.'s work for Puppet was replicated for Ansible and Chef [30]. Schwarz et al.'s study was focused on Chef but was based on a previous work on Puppet [38]. Although approaches based on machine learning to detect faulty scripts have been applied to Puppet [31,32], to the best of our knowledge only Palma et al. applied novelty detection to IaC scripts and, specifically, only to Ansible scripts. Lastly, if possible, we would like to implement a similar analysis as the one implemented in Rehearsal to check determinism and idempotence [37]. Although Sotiropoulos et al. [40] developed a tool which is more effective and without limitations in the manifests analyzed (Rehearsal can not correctly analyze scripts with the *exec* resource), Rehearsal is the only **static** analysis tool we found for this purpose. However, further investigation is necessary to verify if this type of analysis makes sense in other IaC tools besides Puppet, since only Puppet allows the definition of dependencies between resources. Even if specific to Puppet, the support of technology specific analyses should be included into our framework. For instance, if more technologies are included, it will be expected that some other analyses will only be applicable to a subset of technologies. We also consider important that researchers can develop their analyses in our framework, even if they only want to support a single technology.

3.4 Intermediate representation

As stated in Section 3.3, the creation of our model is based on the analysis techniques we want to support. We decided to create a model from scratch because we did not find a model capable of enabling the generalization of the analysis techniques we chose. We could have decided to extend existent representations in the IaC domain, namely the intermediate languages to describe filesystem manipulations mentioned in Section 2.3. However, these representations are not able to capture essential concepts such as resource attributes, which are necessary for the analysis of security smells presented in the work of Rahman et al. [29]. Although we do not consider these filesystem manipulation languages

in the current work, in the future we may extend our representation with the intermediate language presented by Shambaugh et al. [37], if the time constraints of our project allow us to implement the analysis techniques in their work.

Even though our model may need to be adapted as work progresses, Figure 5 describes components of the intermediate representation we propose. Figure 9, in Appendix A, shows the first version of the abstract syntax of our intermediate representation. The syntax will be explored in more detail in the next phase of our project. To implement the work of Rahman et al. [29], we need information about components of scripts such as attributes, variables, and comments. For instance, in SLIC, the tool the authors developed, the rule to find usage of HTTP without TLS first checks if the component is an attribute or a variable, and then verifies if the component's value contains the string *"http"*. The rule to find suspicious comments checks for words such as *"bug"* or *"hack"* in the content of comments. Schwarz et al. [36] require the knowledge about the order and number of attributes in atomic units to check for the smells *Long Resource* and *Misplaced Attribute*, respectively. This information is encoded in our representation by the child-parent relationship between attributes and atomic units. *Unstructured Module* is a code smell detected by checking if a module has the expected file organization, for instance, if a Chef cookbook has a folder for recipes. The file organization of modules is present in our intermediate model by representing what directories and files a module includes. Unit blocks' dependencies are used for code smells *Weakened Modularity*, *Law of Demeter* and *Include Consistency*. Palma et al's work [19] will use the code metrics explained in Section 3.5.

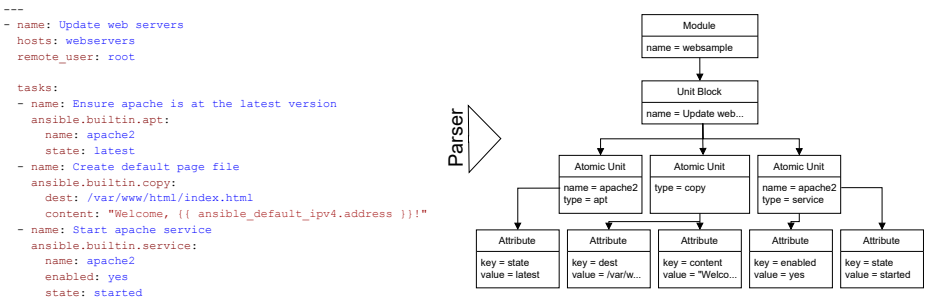

Fig. 6. Example of the parsing of an Ansible script (1) to our intermediate representation.

Figures 6 and 7 describe the parsing to our intermediate representation of the examples in figures 1 and 3. We can see that our model generates a tree of relationships between IaC components. We also show in these figures that the nodes of our representation can carry information about themselves, such as, the type of atomic units. The source code of a component is an example of information to be saved in each node. The two scripts that are transformed

in the figures have very similar functionality and generate similar intermediate representations. The differences appearing in the parsers' outputs are related to specificities in each technology. In this phase of our work and considering the analysis we proposed, we do not consider these differences will cause issues in the end results. However, we could imagine a scenario where, for instance, the type of the atomic unit is relevant for the analysis. In this case, although there is a one by one correspondence between the atomic units' functionality of the Ansible and Chef scripts, the output of the analysis would be different. To create a more uniform representation between technologies, we could map Ansible modules to types of Chef resources. For example, the *apt* and *yum* modules could be mapped to the *package* resource type. Approaches like this one may be considered throughout the development of our project if needed.

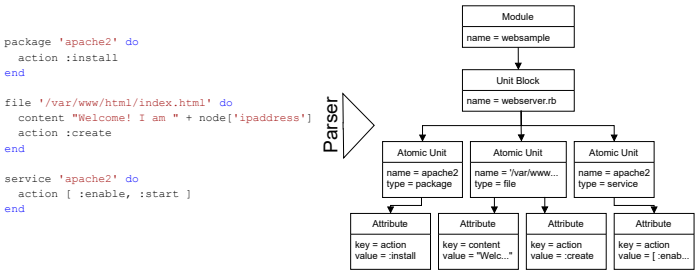

Fig. 7. Example of the parsing of a Chef script (3) to our intermediate representation.

3.5 Code metrics

As described in Section 2.2.1, approaches to identify faulty scripts mainly use code metrics as data to train machine learning methods. Namely, Palma et al. [19] use metrics such as average size of atomic units, number of comments per script and number of lines of code per script. The code metrics component of our architecture is responsible to collect this type of information from the intermediate representation. As we will describe in Section 3.6, we will follow a object-oriented approach to develop our framework. Figure 8 shows how metrics will be applied in this approach. We use a design pattern, called Visitor [7], to allow the creation of new metrics without having to change code of the classes representing the components of the intermediate representation (e.g. Module). In this way, we can easily extend our set of code metrics. If we wanted to create a new metric to count the number of lines in any component, we would only need to create a new class which would implement the method *compute* for each type of component. Components' objects should contain their source code because this will enable the computation of multiple metrics (e.g. number of lines of code) without having to save specific information needed by these metrics in the objects (e.g. saving the number of lines of code in a field). The analysis techniques will use the code metrics in combination with the intermediate representation.

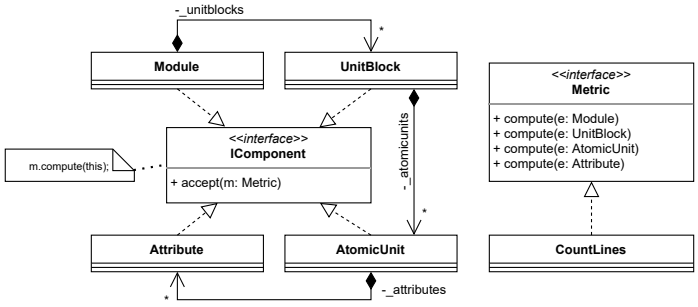

Fig. 8. UML describing the object oriented approach to calculate the code metrics based on the intermediate representation.

3.6 Technical execution

Besides the architectural aspects of our solution, we also consider the implementation to be important. We will follow a object-oriented approach to develop our framework. The programming language we will choose is Python since it is one of the most popular programming language [16], it is object-oriented and has one of the most popular packages for machine learning, *scikit-learn*[13], which is the package used by Palma et al. in their work [19]. To parse Ansible scripts we will use the *yaml* package[14] for Python. To parse Chef and Puppet scripts, we might need to write our own parser, since to the date of writing we were not able to find any promising parsers to scripts of these technologies. Finally, we would like to explore the possibility of delivering our tool, created from developing analyses in our framework, as a service, namely as a GitHub bot. Sadowski et al. mention the importance of including detection tools in the developers' workflow, namely at the moment of code review, and present the successes they obtained by implementing this approach at Google [35].

4 Evaluation and Work Schedule

To test the results of the tool we will develop in our framework, we will use the benchmarks of the original papers for each technique and, if the benchmarks are not available for all the technologies, we will replicate them to those technologies.

For the detection of security smells by Rahman et al. [29], the authors provide a dataset for Puppet [25], and datasets for Ansible and Chef [26] were built in the replication study conducted by Rahman et al. [30]. We will use these datasets to check the accuracy of our tool with respect to precision and recall as the authors did for their tools (SLIC and SLAC). We will then compare the results of our tool with the results in the original work to verify the value of our approach.

Schwarz et al in their work [36] did not calculate accuracy measures. The authors only did tests with small samples of at least 10 occurrences per smell

[13] `https://scikit-learn.org` [14] `https://github.com/yaml/pyyaml`

to decrease the number of false positives. The focus was on the distribution of types of smells in the scripts and the comparison of this distribution for Chef with the distribution for Puppet found by Sharma et al. [38]. We will try to verify if our tool is able to obtain similar distributions to those found by the mentioned authors for our 3 technologies supported. We will use the dataset created by Sharma et al. for Puppet[15] to extract the repositories used, run our tool on these repositories, and check if we obtain similar results. Unfortunately, we were not able to find the datasets used by Schwarz et al. and so we will have to create datasets for both Ansible and Chef. To build these datasets we will mine OSS projects from GitHub using the criteria suggested by Rahman et al. [29]:

1. At least 11% of the files belonging to the repository must be IaC scripts.
2. The repository is not a clone.
3. The repository must have at least two commits per month.
4. The repository has at least 10 contributors.

After we have the datasets, we will run our tool on them, and check the results of the code smell detection technique. We will also run the tool created by Schwarz et al. [36] on the Chef dataset to allow a more accurate comparison with our tool. Finally, if the time constraints allow us, we would like to check accuracy measures for this technique. To do this, we would do qualitative analysis (inspired on Rahman et al's work [29]) to a small subset of each dataset. We would then check the accuracy by comparing the results of our tool with the manual inspection.

To evaluate the generalization of the novelty detection technique to detect faulty scripts by Palma et al. [19], we will run our tool on the dataset used in their work [17], and we will compare the results between the two tools for the Ansible technology. We will then use the same criteria used by Palma to build the dataset for Ansible to build datasets for Chef and Puppet. We will run our tool in these datasets and check if the results are similar to the ones with Ansible. If so, not only the technique is generalizable, but our framework also has the right abstractions to allow the generalization.

A Gantt chart that describes our work schedule is present in Appendix A, namely in Figure 10. We will start by implementing a first prototype capable of generalizing the analysis of security smells [29]. This first prototype might only support Ansible and Chef. We will describe it in a scientific paper which we will try to submit to the ASE conference[16] until the deadline on 29th April 2022. After that, we will make two more iterations of implementation and evaluation for the two other analyses techniques we want to implement, while writing the alpha version of the dissertation. Finally, if the plan goes accordingly to the expected, we will try to implement the extra work we mentioned throughout this document. Namely, we will try to implement and evaluate the analysis to check determinism and idempotence developed by Shambaugh et al. [37], imple-

[15] `https://github.com/tushartushar/configSmellData`
[16] `https://conf.researchr.org/home/ase-2022`

ment the pipeline of techniques into our framework, and possibly doing some experiments to check the impact of this approach.

5 Conclusion

The interest in Infrastructure as Code (IaC) has increased steadily since 2015 [28]. Practitioners adopt IaC because of its benefits, such as, decreasing maintenance costs and reducing the risks of manual misconfiguration by human error. However, IaC suffers from the same problems as traditional software engineering, namely, bugs in scripts. In this document, we describe multiple analysis approaches which can mitigate the lifespan of such errors. For instance, Rahman et al. [29] applied qualitative analysis to scripts in order to identify security smells. The identified smells can be detected through simple string matching rules applied to IaC scripts. By analysing these approaches, we discovered that they all have a problem in common; they are implemented only to one technology. Even though the tools are technology specific, we found the industry to use multiple IaC technologies, even for the same purpose [8]. Having this into consideration, we proposed a technology agnostic framework on top of which static analysis can be developed. The framework uses an intermediate representation to which IaC scripts from multiple technologies are translated to. The static analyses are executed with this intermediate representation as input. We describe the application of intermediate representations to similar problems in other areas of Computer Science to justify the success we expect from our approach. Three analysis techniques were selected to be implemented with the goal of testing our framework: detection of security smells [29], code smells [36], and faulty scripts [19]. We will compare the results we obtain with the techniques implemented in our framework to the results of the original works to evaluate our approach.

References

1. Alnafessah, A., Gias, A.U., Wang, R., Zhu, L., Casale, G., Filieri, A.: Quality-aware devops research: Where do we stand? IEEE Access **9**, 44476–44489 (2021)
2. Ansible, Inc.: Ansible Documentation (2021), `https://docs.ansible.com/`, visited on 2021-12-27
3. Borovits, N., Kumara, I., Krishnan, P., Palma, S.D., Di Nucci, D., Palomba, F., Tamburri, D.A., van den Heuvel, W.J.: Deepiac: deep learning-based linguistic anti-pattern detection in iac. In: Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation. pp. 7–12 (2020)
4. Bruce, S.: Ansible vs Chef: Which configuration management tool is best? (Dec 2020), `https://careerkarma.com/blog/ansible-vs-chef/`
5. Chen, W., Wu, G., Wei, J.: An approach to identifying error patterns for infrastructure as code. In: 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). pp. 124–129. IEEE (2018)
6. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: European symposium on programming. pp. 125–128. Springer (2013)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J., Patterns, D.: Elements of reusable object-oriented software, vol. 99. Addison-Wesley Reading, Massachusetts (1995)
8. Guerriero, M., Garriga, M., Tamburri, D.A., Palomba, F.: Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 580–589. IEEE (2019)
9. Ikeshita, K., Ishikawa, F., Honiden, S.: Test suite reduction in idempotence testing of infrastructure as code. In: International Conference on Tests and Proofs. pp. 98–115. Springer (2017)
10. Janardhan, S.: More details about the october 4 outage (Oct 2021), `https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/`
11. Leino, K.R.M.: This is boogie 2. manuscript KRML **178**(131), 9 (2008)
12. Leino, K.R.M.: Program proving using intermediate verification languages (ivls) like boogie and why3. In: Proceedings of the 2012 ACM conference on High integrity language technology. pp. 25–26 (2012)
13. Lepiller, J., Piskac, R., Schäf, M., Santolucito, M.: Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities. In: TACAS (2). pp. 105–123 (2021)
14. Li, H., Zhang, P., Sun, G., Hu, C., Shan, D., Pan, T., Fu, Q.: An intermediate representation for network programming languages. In: 4th Asia-Pacific Workshop on Networking. pp. 1–7 (2020)
15. Monperrus, M., Urli, S., Durieux, T., Martinez, M., Baudry, B., Seinturier, L.: Repairnator patches programs automatically. Ubiquity **2019**(July), 1–12 (2019)
16. O'Grady, S.: The RedMonk Programming Language Rankings: June 2021 (08 2021), `https://redmonk.com/sogrady/2021/08/05/language-rankings-6-21/`
17. Palma, S.D.: Defect prediction tool validation dataset 1 (Jun 2020). https://doi.org/10.5281/zenodo.3906023
18. Palma, S.D., Di Nucci, D., Palomba, F., Tamburri, D.A.: Within-project defect prediction of infrastructure-as-code using product and process metrics. IEEE Transactions on Software Engineering pp. 1–1 (2021)
19. Palma, S.D., Mohammadi, M., Di Nucci, D., Tamburri, D.A.: Singling the odd ones out: a novelty detection approach to find defects in infrastructure-as-code. In: Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation. pp. 31–36 (2020)

20. Potvin, R., Levenberg, J.: Why google stores billions of lines of code in a single repository. Communications of the ACM **59**(7), 78–87 (2016)
21. Progress Chef: Case Study - A Lesson in Digital transformation in the Midst of a Global Pandemic (2020), https://www.chef.io/customers/edgenuity
22. Progress Chef: Chef Documentation (2021), https://docs.chef.io/, visited on 2021-12-29
23. Puppet Labs: Puppet Documentation (2021), https://puppet.com/docs, visited on 2021-12-29
24. Puppet Labs, Zivra: Case Study - Jewelers Mutual Insurance Company collaborated with Zivra to speed up the creation of environments for maximum efficiency (2019), https://puppet.com/resources/customer-story/jewelers-mutual
25. Rahman, A.: The Seven Sins: Security Smells in Infrastructure as Code Scripts (Jan 2019). https://doi.org/10.6084/m9.figshare.6943316.v4, https://figshare.com/articles/dataset/Dataset_The_Seven_Sins-Security_Smells_in_Infrastructure_as_Code_Scripts/6943316/4
26. Rahman, A.: Dataset for 'Security Smells for Ansible and Chef Scripts: A Replication Study' (Apr 2020). https://doi.org/10.6084/m9.figshare.8085755.v2, https://figshare.com/articles/dataset/Dataset_for_Security_Smells_for_Ansible_and_Chef_Scripts_A_Replication_Study_/8085755/2
27. Rahman, A., Farhana, E., Parnin, C., Williams, L.: Gang of eight: A defect taxonomy for infrastructure as code scripts. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). pp. 752–764. IEEE (2020)
28. Rahman, A., Mahdavi-Hezaveh, R., Williams, L.: A systematic mapping study of infrastructure as code research. Information and Software Technology **108**, 65–77 (2019)
29. Rahman, A., Parnin, C., Williams, L.: The seven sins: Security smells in infrastructure as code scripts. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 164–175. IEEE (2019)
30. Rahman, A., Rahman, M.R., Parnin, C., Williams, L.: Security smells in ansible and chef scripts: A replication study. ACM Transactions on Software Engineering and Methodology (TOSEM) **30**(1), 1–31 (2021)
31. Rahman, A., Williams, L.: Characterizing defective configuration scripts used for continuous deployment. In: 2018 IEEE 11th International conference on software testing, verification and validation (ICST). pp. 34–45. IEEE (2018)
32. Rahman, A., Williams, L.: Source code properties of defective infrastructure as code scripts. Information and Software Technology **112**, 148–163 (2019)
33. Red Hat: Customer Case Study - German Federal Office speeds I.T. management by 50% with Red Hat Ansible Tower (2018), https://www.ansible.com/hubfs/Images/resources/rh-ble-case-study-f13497wg-201810-en_1.pdf?hsLang=en-us
34. Rockwell, N.: Summary of June 8 outage (Jun 2021), https://www.fastly.com/blog/summary-of-june-8-outage
35. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C.: Lessons from building static analysis tools at google. Communications of the ACM **61**(4), 58–66 (2018)
36. 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). pp. 220–228. IEEE (2018)
37. Shambaugh, R., Weiss, A., Guha, A.: Rehearsal: A configuration verification tool for puppet. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 416–430 (2016)

38. Sharma, T., Fragkoulis, M., Spinellis, D.: Does your configuration code smell? In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR). pp. 189–200. IEEE (2016)
39. Silva, D., Silva, J., Santos, G.J.D.S., Terra, R., Valente, M.T.O.: Refdiff 2.0: A multi-language refactoring detection tool. IEEE Transactions on Software Engineering (2020)
40. Sotiropoulos, T., Mitropoulos, D., Spinellis, D.: Practical fault detection in Puppet programs. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 26–37 (2020)
41. Weiss, A., Guha, A., Brun, Y.: Tortoise: Interactive system configuration repair. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 625–636. IEEE (2017)

A Appendix

```
Module          <module> ::= Module{name: <str>, blocks: [<block>*],
                     dirs: <directory>*, files: <file>*}
Unit Block      <block> ::= UnitBlock{name: <str>, vars: [<variable>*],
                     units: <unit>*, comms: <comment>*, dependencies: <block>*}
Atomic Unit     <unit> ::= Unit{name: <str>, attrs: <attribute>*}
Attribute       <attribute> ::= Attribute{name: <id>, value: <value>}
Variable        <variable> ::= Variable{name: <id>, value: <value>}
Comment         <comment> ::= Comment{content: <str>}
Directory       <directory> ::= Directory{name: <str>, path: <str>, files: <file>*}
File            <file> ::= File{name: <str>, path: <str>}
                <value> ::= <str> |
                    <number> |
                    <value>* |
                    <id>
                <id> ::= ;sequence of alphanumerics which starts with a letter
                <str> ::= "<character>*"
                <number> ::= ;integer or double
```

Fig. 9. First version of the abstract syntax of our intermediate representation.

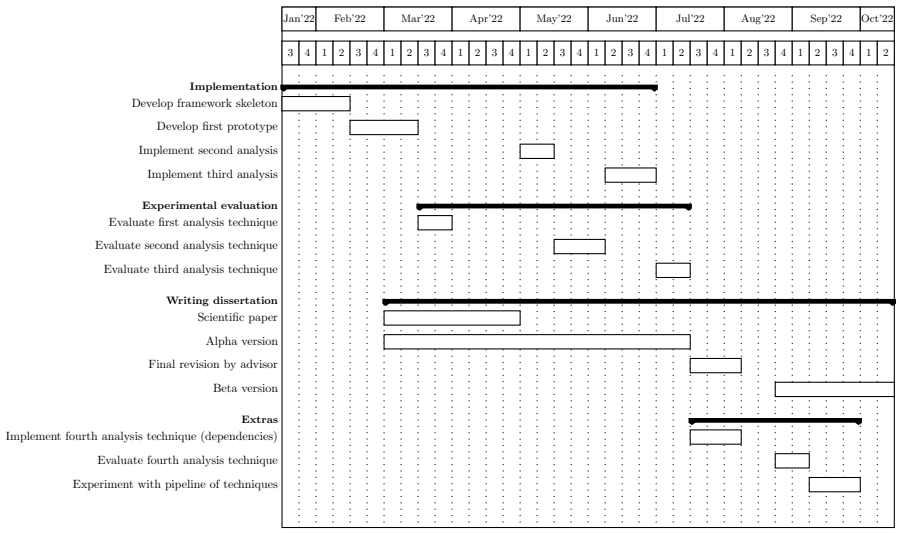

Fig. 10. Planned Schedule