university of groningen

faculty of science and engineering

Catching Cost Issues in Infrastructure as Code Artifacts using Linters

Koen Bolhuis

university of groningen

faculty of science and engineering

University of Groningen

Catching Cost Issues in Infrastructure as Code Artifacts using Linters

Master's Thesis

To fulfill the requirements for the degree of
Master of Science in Computing Science
at the University of Groningen under the supervision of
Prof. dr. V. (Vasilios) Andrikopoulos (Bernoulli Institute, University of Groningen)
and
dr. D. (Daniel) Feitosa (Bernoulli Institute, University of Groningen)

Koen Bolhuis (S3167895)

August 1st, 2024

Abstract

Cost concerns have historically been a driving factor in cloud adoption. As the complexity of cloud-based software systems grows, and with it the need to manage increasingly intricate infrastructures, Infrastructure as Code (IaC) approaches have become indispensable tools. However, few studies have examined the cost implications of IaC usage for cloud software. In this research, we use an existing dataset that analyzed cost-related commits on IaC artifacts from open-source repositories. We apply thematic analysis to the commits' contents to identify recurring effective and ineffective practices and we compile a catalog of cost management patterns and antipatterns. This catalog can serve as a foundation for improving cost-efficiency, but to foster adoption it would be beneficial to incorporate the patterns and antipatterns directly into the development toolchain. Since static analysis tools such as linters are widely used to improve non-functional properties and catch issues in IaC scripts, and because existing tools both in literature and industry focus on security and code quality concerns as opposed to cost management, we implement selected (anti)patterns as rules in two popular linters, Checkov and TFLint, to aid developers in cost-effective IaC development.

Acknowledgements

My sincere thanks go out to everyone who has helped bring this thesis to life. First of all, I want to extend my gratitude to Vasilios Andrikopoulos and Daniel Feitosa, for their fantastic guidance and collaboration, and for the opportunity to work on this project. To my family, for their unconditional love and support. To my friends, for the days of "working" together in the study landscape, and the necessary distractions from the thesis. Thanks as well to the other students doing their projects under the cloud cost awareness umbrella, who made this endeavour feel much less like a solo journey.

Without all of you, this thesis would not have been possible—so thank you.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Since its introduction, cloud computing has become a wildly popular way to implement computational infrastructures. The flexibility offered by cloud technologies, among other factors, has driven their widespread adoption by organizations, but one key driver stands out in particular: the potential of cost reduction. By removing the need for up-front capital expenses in favor of utilities-like, pay-as-you-go pricing [3, 61], and by benefitting from the economies of scale achieved by cloud service providers, businesses and individuals alike are able to deploy larger and more complex infrastructures at manageable prices.

However, this increasing infrastructural complexity needs to be managed somehow. Manual configuration and deployment is labor-intensive and susceptible to errors [91], a challenge which is only exacerbated by the growing demand for multi-region, multi-cloud deployments, and so a need for automation arises [10, 29]. Infrastructure as Code (IaC) solutions, particularly *infrastructure orchestrators* such as Terraform and Amazon Web Services (AWS) CloudFormation, have emerged to address this need by adding a layer of abstraction over cloud providers' management APIs [29] and by enabling developers to provision their infrastructures through reusable artifacts that are treated like any other type of source code [8].

Previous research by Feitosa et al. [33] has already uncovered evidence of cost-related decision making in the version control systems of IaC-based open-source software and provided evidence for developers' awareness of cost at the level of Infrastructure as Code. Nevertheless, little other IaC research has touched on these cost concerns, instead focusing on security and quality aspects of IaC scripts and tooling to support IaC practices [42, 76, 83].

This thesis aims to address this gap in two steps. First, we build upon an existing set of commits that resulted from Feitosa et al.'s work [34] by means of repository mining and thematic analysis. We carefully analyze how developers address existing cost issues in the IaC artifacts involved in these commits to extract effective and ineffective cost management practices, and we present these practices in the form of a collection of patterns and antipatterns. Patterns are a concept originally introduced in the context of object-oriented design [40] which present reusable solutions to common software development problems. Similarly, antipatterns represent recurring pitfalls that can serve to educate developers of issues that need to be avoided [17].

Secondly, we transform selected patterns and antipatterns into detection rules, similar to previous work [81, 86]. We then transfer these rules to a type of automated static analysis tools called linter to facilitate a “left shift” of cost management within the development process of systems that deploy Infrastructure as Code. Linters tend to employ relatively simple rules to detect problems and defects like code smells [97], but they are also suitable to catch the cost issues that are the focus of this work. They integrate with development workflows by pointing out the exact location of issues, for example inside an integrated development environment (IDE). Currently, no existing IaC linters detect cost issues; thus, we extend two popular linters for code and security smells.

1.1 Research Objective

In short, our objective is to help developers better manage the cost of their Infrastructure as Code-enabled systems. To achieve this goal, we aim to implement a linter that automatically detects common cost issues. We define the following research questions to guide this process:

RQ_1 What recurring patterns can we find in code changes that address cost issues in the IaC artifacts of cloud software?

RQ_2 How can these patterns be implemented in a linter?

RQ_3 How well does the resulting linter perform at detecting cost issues?

Through **RQ_1**, we aim to extract general patterns from a set of code changes where the message attached to each commit has previously been determined to be related to cost. In this way, we want to find out how practitioners are addressing cost concerns in their Terraform codebases. With **RQ_2**, we build on the aforementioned patterns by transforming them into rules that can be implemented in a code linter, in order to automate the detection of instances of these patterns. Finally, through **RQ_3**, we determine how well the implementation performs at detecting cost issues, as a proxy for how useful the linter may be for developers of IaC-based cloud software.

1.2 Contributions

The main contributions of this work can be summarized as follows:

1. A labeled dataset of commits and the cost-changing actions occurring in their diffs;
2. A catalog of cost management patterns and antipatterns for IaC;
3. An implementation of selected (anti)patterns as Checkov [1] checks and a TFLint [2] ruleset.

Supplementary code and data for this thesis are available in a separate package [12].

Furthermore, as a side artifact of our literature search we provide an overview of static analysis tools for IaC from literature and industry. In addition, through the evaluation of our implementation we find several unaddressed cost issues in open-source repositories, which may provide opportunities for future investigation.

[1] https://checkov.io
[2] https://github.com/terraform-linters/tflint

1.3 Outline

The rest of this thesis is organized as follows. Chapter 2 introduces the necessary background and surveys literature related to IaC, linters, code smells and the combination thereof. Chapter 3 specifies requirements for the linter and our approach towards implementing it. Next, Chapter 4 consists of a thematic analysis of IaC code changes to extract recurring cost patterns and antipatterns. These are then used in Chapter 5 to implement linter rules to detect cost issues in IaC files, followed by an evaluation in Chapter 6. Finally, we discuss our findings in Chapter 7 and conclude the thesis in Chapter 8.

2 Background & Related Work

In this chapter, we look at a number of broad topics, the intersection of which forms the basis of our work: *code smell detection*, *linters* and *Infrastructure as Code*, with a particular focus on linters and static analyzers for the latter. For our literature search, we use the following search queries on Google Scholar:

- (program OR code OR software) AND lint*
- code smell AND detect*
- ”infrastructure as code” AND (smell OR defect OR cost OR energy OR static analysis OR lint*)

For tools used in industry, we search Google using the following queries:

- infrastructure as code linter
- infrastructure as code analyzer
- IaC linter
- IaC analyzer
- terraform linter
- terraform analyzer

We explicitly search for Terraform-oriented tools because the dataset by Feitosa et al. [34] that serves as the starting point for our study is focused on Terraform, making those tools especially relevant.

2.1 Code Smell Detection

The term “code smell”, coined by Beck and popularized by Fowler [39], refers to defects and flaws in code that are not necessarily coding bugs or errors by themselves, but may be indicators of deeper issues. The exact definition of what constitutes a code smell is subjective and varies by domain, language and developer. Since the concept was introduced, it has been extended to many programming languages and domains, including Infrastructure as Code; we discuss this in more detail in Section 2.4. The issues we investigate can be considered a type of “cost smell”, and so techniques for code smell detection are of potential interest. As noted by Santos et al. [87] in their systematic review on code smells, tools and methods for code smell detection are a key area in code smell research.

Schumacher et al. [88] studied professional software developers’ ability to detect code smells compared to automated methods and found that automated methods performed better than humans, who often tended to disagree with one another, suggesting that automated detection as a first step in code review can decrease the effort spent on manual code inspections. The low agreement among developers is also apparent from the work by Hozano et al. [49], who studied how developers detect code smells and found that individual developers do so in significantly different ways. Both studies support the benefits of automated tools to detect issues in code.

One category of detection tools consists of *metric-based* approaches. Danphitsanuphan and Suwantada [28] implemented code smell detection based on code metrics, as well as an investigation into the correlation of these smells with structural bugs. Velioglu and Selcuk [102] also introduced a metric-based approach where a set of training projects is used to determine lower and upper bounds for code smell metrics to detect antipatterns. Arcelli Fontana et al. [36] proposed a benchmark-based approach to automatically derive threshold values for metric-based code smell detection. JSNOSE, introduced by Fard and Mesbah [32], combines static and dynamic analysis of JavaScript code using a metric-based approach. A limitation shared by these techniques that use code metrics is that they are often unable to highlight the exact location and source of issues, because they rely on aggregate statistics to determine whether some higher-level unit of code, e.g. class or file, contains code smells.

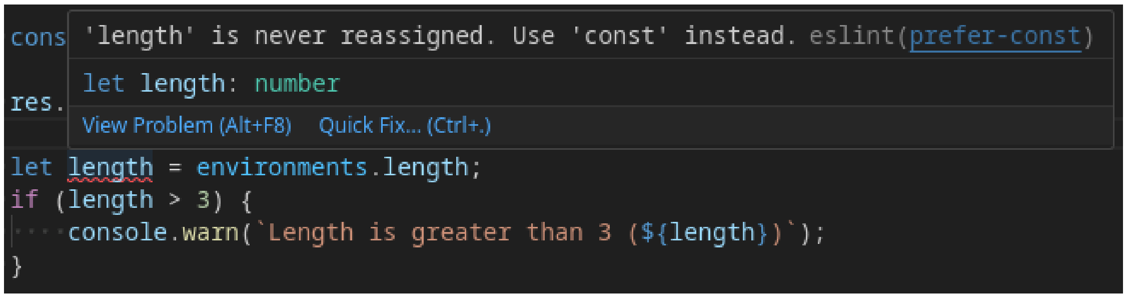
In addition to metric-based methods, various other approaches have been proposed. Moha et al. [62] introduced DECOR, a method consisting of a set of tools to specify design smells and their underlying code smells, and detect them using code generation from a rule-based domain-specific language combining metrics, relations and other rules. Rasool and Ali [82] specified Android-specific code smells and implemented an AST-walking tool to detect them. Another tool for Android smell detection is the aDoctor project by Palomba et al. [72], which uses a combination of pattern matching and graph analysis. Walker, Das and Cerny [103] studied the automated detection of microservice-specific code smells by using a combination of graph analysis and metrics on dependency and configuration artifacts. Despite the variety in approaches, a common theme among these studies is the fact that they encode code smells as rules which can detect instances of these smells in source code, which is a simple but flexible way to implement code smell detection.

Recent efforts have focused on *machine learning* (ML) methods for code smell detection. MLSmellHound by Kannan et al. [55] uses ML to adapt Pylint [1] results to include relevant context around the detected smells. Fontana et al. [37], Dewangan et al. [30] and Liu et al. [60] each proposed approaches for code smell categorization based on ML and deep learning. Di Nucci et al. [31] performed an empirical evaluation of ML-based smell detectors, concluding that existing tools performed poorly on codebases containing mixed types of code smells. Pecorelli et al. [73] compared ML methods with DECOR [62] and found that while DECOR generally performs better than the machine learning methods, its precision is still too low for practical use. Similar to metric-based methods, ML methods tend to focus on categorization of smells as opposed to identifying the exact code location, significantly limiting their utility for practical static analysis tools. All in all, ML approaches for smell detection still have a ways to go.

[1] https://www.pylint.org/

2.2 Linters

Automatic static analysis tools (ASATs) enable their users to inspect code and find problems like defects, style issues and deviations from best practices. ASATs aid in detecting faults and highlighting refactoring opportunities early in the software development life cycle, when they require less effort to address and are cheaper to fix, which has made them popular among development teams [51]. A linter is a type of static analysis tool that tends to perform relatively simple types of analyses to catch low-complexity issues such as code smells or coding style violations [96]. Linters often integrate with developers' code editors, allowing them to point at the exact location of issues in the code the moment these issues are introduced. Examples of popular linters include Pylint for Python and ESLint [2] for JavaScript. Figure 2.1 shows an example of how a linter commonly presents issues to developers.



```
cons  'length' is never reassigned. Use 'const' instead. eslint(prefer-const)
      let length: number
res.
      View Problem (Alt+F8)   Quick Fix... (Ctrl+.)
let length = environments.length;
if (length > 3) {
   console.warn(`Length is greater than 3 (${length})`);
}
```

Figure 2.1: Example linter warning from ESLint in Visual Studio Code

To our knowledge, no secondary study exists that specifically covers linters. However, there have been empirical studies into the use of linters in industry and works that introduce new linters. For example, Hericko and Sumak [48] performed an MSR study to measure linter usage and warnings in the JavaScript open-source ecosystem. Tomasdottir, Aniche and Van Deursen [97] interviewed 15 developers of projects that use ESLint on why and how they use the linter. They extended this research by also analyzing over 9 500 ESLint configurations and performing a survey among 337 JavaScript developers on linter use [96]. Habchi, Blanc and Rouvoy [44] interviewed 14 developers on the use of linters to detect performance issues in Android applications. These studies all highlight the benefits that developers experience when adopting linters in their projects, which include catching issues earlier in the development cycle, preventing errors, and simplifying code reviews. However, developers also face challenges, such as creating and maintaining the linters' configuration, choosing which rules to enable or disable, and dealing with false positives, as well as the fact that many linter rules are based on experience as opposed to real-world evidence.

A number of studies have themselves introduced linters or extensions to existing linters. The original linter was Johnson's Lint [54], released in 1978, which provided C programmers with analyses of C programs that went beyond those offered by C compilers of the time. More recently, Rafnsson et al. [74] created a plugin for ESLint which can automatically detect cross-site scripting, security misconfigurations and SQL injections using rules that walk JavaScript abstract syntax trees (ASTs). Goaer [41] introduced an

[2] https://eslint.org

extension to Android Lint based on visitor-style rules to detect Android-specific energy efficiency bugs. Ryou et al. [85] created Culint, a tool that uses two fine-tuned language models to respectively classify variable-misuses in Python functions and suggest fixes.

DevReplay by Ueda, Ishio and Matsumoto [99] goes a step further by analyzing regular programming behavior on open-source projects to automatically generate regular expression rules that can be used to detect project-specific coding rule violations. Vassallo et al. [101] introduced CD-Linter, a linter for GitLab continuous integration/continuous delivery (CI/CD) pipelines in Maven or Python projects, using pattern matching on relevant artifacts to detect "CD smells". Meanwhile, Sprinter by Alfredo, Santos and Garrido [1] is a linter for Java that uses control-flow analysis and pattern matching on control-flow graphs to detect structured programming issues. Almashfi and Lu [2] created TAJS$_{\text{lint}}$ for JavaScript, which uses a combination of AST-walking rules and control-flow analysis to detect JavaScript-specific code smells. It is clear that while there are many different approaches to implementing linters, they generally have one thing in common: the use of relatively simple rules to perform the detection of their respective issues in source code.

2.3 Infrastructure as Code

Infrastructure as Code is a collection of techniques where the infrastructure of a software system is deployed and configured using code, as opposed to manual configuration by system administrators using e.g. interactive installation tools. Popularized as part of the DevOps movement, IaC promotes the creation of reusable scripts to manage infrastructure, and it represents a widely-used practice [8, 63]. IaC is applied to various facets of infrastructure management, such as *configuration management* across local and remote machines and *infrastructure orchestration* of cloud service provider resources. Popular examples of the former include Puppet [3], Chef [4] and Ansible [5], while the latter category includes tools such as AWS CloudFormation [6] and Terraform [7], the focus of this study.

To illustrate how Terraform can be used to manage cloud resources, Listing 2.1 contains an example of a Terraform configuration, written in the HashiCorp Configuration Language (HCL) [8]. Terraform uses so-called *providers* to abstract away the management APIs offered by cloud providers. The primary construct in Terraform is the `resource` block, which allows the developer to declare cloud resources and their properties. In the example, a virtual server is defined with properties like the machine image (`ami`) and instance type. The language also supports additional constructs such as variables, loops and external modules, in order to manage complexity and abstract the creation of similar resources. When Terraform is run using a configuration as its input, it uses the cloud provider's API to compare the actual state of the cloud infrastructure to the desired state, and it creates a plan to perform the necessary changes that achieve this

[3]https://puppet.com
[4]https://chef.io
[5]https://www.ansible.com/
[6]https://docs.aws.amazon.com/cloudformation/
[7]https://terraform.io
[8]https://developer.hashicorp.com/terraform/language

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }

  required_version = ">= 1.2.0"
}

resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t3.micro"
}
```

Listing 2.1: Terraform configuration that provisions a compute instance on Amazon Web Services, adapted from the Terraform tutorial [9]

desired state. This plan can then be executed to apply the modifications or create new resources.

Despite the fact that IaC is undeniably a cornerstone of cloud-based software development, it is still a rather new practice, something which is also reflected by the relatively sparse literature on the subject. In a 2019 systematic mapping study, Rahman, Mahdavi-Hezaveh and Williams [76] identified the focus of existing studies on tools or extensions of tools implementing IaC practices, noting a lack of research into defects and security flaws. This was partly corroborated by a study from Guerriero et al. [42], who interviewed 44 senior developers and found that support provided by existing tools is still limited, and that there is a need for novel techniques for testing and maintaining IaC.

After those studies were published, more research efforts were directed towards quality and security concerns. Kumara et al. [57] performed a grey literature review to find 10 categories of good practices and 4 categories of bad practices for IaC, specifically for the IaC tools Ansible, Puppet and Chef, while Chiari, De Pascalis and Pradella [21] and Reddy Konala, Kumar and Bainbridge [83] reviewed the landscape of IaC static analysis tools, finding a large number of tools focused on security and code quality. Hasan, Bhuiyan and Rahman [45] examined internet artifacts such as blog posts to extract prevalent testing practices for IaC, the most notable ones being the use of linters for the avoidance of antipatterns and the use of continuous integration to validate changes. Dalla Palma et al. [26] compiled a catalog of 46 metrics for measuring the quality of IaC scripts, which are used in a framework called RADON [71] for code smell and defect prediction. Although all of the aforementioned studies have contributed to the understanding of security and quality aspects of IaC scripts, research is in its infancy, and problems are still wide-spread in practice, as determined by the various studies that have performed empirical analysis of IaC artifacts.

For example, Rahman, Farhana and Williams [75] performed quantitative analysis on open-source IaC scripts combined with practitioner interviews to extract antipatterns

[9] https://developer.hashicorp.com/terraform/tutorials/aws-get-started/aws-build

that correlate with defects, finding a set of 5 antipatterns—primarily related to the development process—that lead to defects. Bhuiyan and Rahman [11] identified security issues in IaC scripts that are frequently co-located, observing that between 17.9% and 32.9% of their inspected scripts contain co-located insecure coding practices.

Using a similar approach to our study, Chen, Wu and Wei [20] looked at code changes where errors in IaC artifacts were fixed, in order to extract common error patterns, based on HDBSCAN clustering. Their study shows the value of extracting issues and defects from code changes, although it should be noted that their sample size of 14 Puppet artifacts is quite small. In another study, Rahman et al. [79] applied orthogonal defect classification [22] to IaC scripts to categorize defects and compare their distribution to non-IaC code. They found that unlike non-IaC codebases, where the frequency of introduced defects is high early in a project's lifecycle but decreases over time, IaC scripts show a consistent temporal trend.

There is, however, very little (if any) literature on the intersection of IaC and cost. As evidenced by the studies discussed so far, most research is concerned with security, defects and quality issues, as well as code smells, which we discuss in more detail in the following section. The lack of focus on cost management is surprising given the fact that cost reduction is a key reason to adopt cloud computing, and by extension IaC, in the first place. The only work we found that is tangentially related to IaC cost concerns is a technical paper by Osaba et al. [69] in the context of the PIACERE project [10] introducing the IaC Optimizer Platform, a tool that can optimize IaC deployments for user-defined constraints on availability, performance and cost. However, the tool is designed to integrate with other tools from the PIACERE ecosystem, preventing its use with existing IaC technologies like Terraform.

2.4 IaC Smells

The concept of (code) smells can be extended to IaC, and numerous studies have looked at code smells in IaC codebases. Guo and Wu [43] performed a systematic literature review on the prevalence and detection of code smells in microservice-based software, including smells in IaC artifacts. Their review revealed a trend towards research into IaC smell detection, but also a lack of studies focusing on impact analysis. Sharma, Fragkoulis and Spinellis [92] analyzed open-source repositories containing Puppet artifacts to extract 13 implementation and 11 design configuration smells. Schwarz, Steffens and Lichter [89] built upon this work by extending the catalog of smells and analyzing to what extent these smells can apply to other IaC technologies, concluding that the original smells can be extended to Chef. Dalla Palma et al. [27] compared machine learning methods for defect prediction in Ansible artifacts. Another study by Dalla Palma, Di Nucci and Tamburri [25] introduced AnsibleMetrics, a tool to compute metrics on Ansible scripts that may predict defects. This tool is used by RADON Framework for IaC Defect Prediction [71], an integrated framework to mine repositories, collect data and train machine learning models for prediction of Ansible defects. Ntentos et al. [66] introduced a method for detecting architectural smells, mainly related to coupling, and for suggesting potential fixes. Rahman and Williams [78] performed qualitative analysis of

[10] https://piacere-project.eu/

IaC scripts involved in defect-related version control commits to extract properties of source code that correlate with defects, and Rahman et al. [80] created a defect taxonomy for IaC. The main thread that runs throughout these existing smell-related studies is their focus on smells relating to *code quality* and *defects*, again emphasizing a distinct lack of focus on the *cost* aspects.

2.5 Static Analysis of IaC

To address code smells and other issues, there is a large number of tools for static analysis of IaC introduced in literature or supported by industry. We provide a summary of the available tools in Table 2.1. We use the table by Reddy Konala, Kumar and Bainbridge [83] as a starting point, and describe the following properties:

- **Issues detected:** the type of issues the tool detects. These include *antipatterns*, *correctness violations*, *code smells* and *security smells*. Additionally, terraform-compliance [70] does not detect any specific kind of issue or even issues per se, instead allowing users to define and detect custom properties.
- **Target(s):** which IaC technology the tool supports. These include *infrastructure orchestrators* such as Terraform, AWS CloudFormation, Azure Resource Manager (ARM) and TOSCA, *configuration management* tools such as Puppet, Chef and Ansible, and *container and image management* tools like Docker, Kubernetes and Helm.
- **Technique:** the detection technique used by the tool. We identify rule-based approaches using *regular expressions* (simple text matches), *ad-hoc rules* (combining ad-hoc properties to detect issues), *graph analysis* (often specifying rules in terms of connections between components) and *deep learning* including a proprietary "AI engine". Moreover, RADON [71] uses data mining and code metrics to detect Ansible antipatterns, Rehearsal [91] uses a SMT solver to encode Puppet scripts and verify certain properties that indicate code smells, and SODALITE [58] uses ontologies and allows users to define SPARQL [11] queries to detect code smells.
- **Extension mechanism:** the method with which the tool can be extended, since this is relevant for our own work. Several tools are proprietary or have not published their source code, preventing them from being extended. Some tools offer dedicated extension or plugin capabilities, while others are open-source but lack these dedicated mechanisms, instead requiring custom extensions to be created.

We have omitted some of the tools from the original table. CloudSploit [5] does not analyze IaC; instead, it connects to cloud providers' APIs to collect information which is then analyzed for security issues; SecGuru by Jayaraman et al. [52] detects issues in firewall policies, which, while they can be considered IaC, are too specialized compared to IaC technologies like Ansible or Terraform; finally, SecureCode by Dai et al. [24] analyzes shell scripts *embedded* in the IaC scripts as opposed to IaC code itself.

In addition, we have extended the set of tools using a combination of Google Scholar and Google search, finding several tools that Reddy Konala, Kumar and Bainbridge did not list: Ansible Lint [4], Bicep linter [100], GASEL [67], Regula [84], terrafirma [104],

[11]https://www.w3.org/TR/sparql11-query/

terraform-compliance [70], terrascan [94], TFLint [95], trivy [7] and an unnamed tool by Opdebeeck, Zerouali and De Roover [68].

Tools which are supported by industry are marked with (I). Tools which can be considered linters are indicated with (L); here, we define linters as tools which perform simplistic forms of analysis and can highlight the exact location of issues. Out of the 31 tools, 15 can be considered a linter under this definition. Because DevOps and specifically IaC promote the use of software engineering practices for infrastructure management [8], we argue that linters, which are popular among regular software development teams, are a suitable way to implement static analysis for IaC. This is compounded by the fact that infrastructure orchestrators directly influence cloud spend through their automated creation of resources, which makes it worthwile to catch potential cost issues as early as possible. However, among the identified tools, none support cost management, a gap we aim to address in this work by implementing a linter for cost issues.

Table 2.1: Tools for static analysis of IaC

Tool	**Issues detected**	**Target(s)**	**Technique**	**Extension mechanism**
ACID [80]	Antipatterns	Puppet	Regular expressions	N/A
Ansible Lint [4] **(I, L)**	Antipatterns	Ansible	Ad-hoc rules	Writing ad-hoc extensions in Python
BARREL [15]	Correctness violations	TOSCA	Graph analysis	Writing rules in JavaScript
Bicep linter [100] **(I, L)**	Code smells, security smells	Bicep	Regular expressions, ad-hoc rules	Writing ad-hoc extensions in C#
Checkov [18] **(I, L)**	Security smells	Terraform HCL/JSON/ plans, CloudFormation, ARM, Kubernetes, Helm, Dockerfile	Regular expressions, ad-hoc rules, graph analysis	Writing rules in YAML or Python
cookstyle [19] **(I, L)**	Code smells	Chef	Regular expressions	Writing RuboCop [12] extensions in Ruby
DeepIaC [14]	Antipatterns	Ansible	Deep learning	N/A
foodcritic [38] **(I, L)**	Code smells	Chef	Regular expressions	Writing rules in a Ruby DSL
GASEL [67]	Security smells	Ansible	Graph analysis	N/A
GLITCH [86]	Security smells	Ansible, Chef, Puppet	Regular expressions	Writing rules and support for additional languages in Python
Häyhä [59]	Security smells	CloudFormation	Graph analysis	N/A
KICS [56] **(I, L)**	Security smells	Terraform HCL, CloudFormation, ARM, Google Deployment Manager, Docker, Docker Compose, Helm, Kubernetes, Knative, Pulumi, Serverless Framework, Ansible	Regular expressions	Writing policies in Rego [13]
Puppeteer [92] **(L)**	Code smells	Puppet	Regular expressions	Implementing detection strategies in Python
RADON [71]	Antipatterns	Ansible	Data mining	Writing ad-hoc extensions in Python
Regula [84] **(I, L)**	Security smells	Terraform HCL/JSON/ plans, CloudFormation, ARM, Kubernetes	Ad-hoc rules, regular expressions	Writing policies in Rego
Rehearsal [91]	Code smells	Puppet	SMT solver	N/A
Semgrep [90] **(I, L)**	Code smells, security smells	Terraform, Dockerfile	Regular expressions	Writing rules in YAML

[12]https://github.com/rubocop/rubocop
[13]https://www.openpolicyagent.org/docs/latest/policy-language/

Tool	**Issues detected**	**Target(s)**	**Technique**	**Extension mechanism**
SLAC [81] **(L)**	Security smells	Ansible, Chef	Ad-hoc rules, regular expressions	Writing rules in Python
SLIC [77] **(L)**	Security smells	Puppet	Ad-hoc rules, regular expressions	Writing rules in Python
Snyk IaC Security [50] **(I)**	Security smells	Terraform HCL, CloudFormation, ARM	Proprietary AI engine	Writing policies in Rego
SODALITE [58]	Code smells	TOSCA	Ontology with SPARQL queries	Writing SPARQL queries for user-defined smells
Sommelier [16] **(L)**	Correctness violations	TOSCA	Ad-hoc rules	Writing rules in Python
SonarLint [93] **(I, L)**	Code smells, security smells	Terraform HCL, ARM, CloudFormation, Kubernetes, Dockerfile	Regular expressions	Writing rules in Java
TAMA [46]	Antipatterns	Ansible	Ad-hoc rules, regular expressions	Writing rules in Python
terrafirma [104] **(I)**	Security smells	Terraform plans	Ad-hoc rules, regular expressions	Writing rules in YAML or Python
terraform-compliance [70] **(I)**	Custom properties	Terraform HCL	Ad-hoc rules	Writing policies using radish BDD [14] constructs
terrascan [94] **(I)**	Security smells	Terraform HCL, CloudFormation, ARM, Kubernetes, Helm, Kustomize, Dockerfile	Ad-hoc rules, regular expressions	Writing policies in Rego
TFLint [95] **(I, L)**	Code smells	Terraform HCL	Ad-hoc rules, regular expressions	Writing rulesets in Go or Rego (experimental)
tfsec [6] **(I, L)**	Security smells	Terraform HCL/CDK	Regular expressions	Writing policies in Rego
trivy [7] **(I, L)**	Code smells, security smells	Terraform HCL/JSON/ tfvars, CloudFormation, ARM, Kubernetes, Dockerfile, Helm YAML	Regular expressions	Writing modules in TinyGo [15]
Opdebeeck, Zerouali and De Roover [68]	Code smells	Ansible	Graph analysis	N/A

[14]https://github.com/radish-bdd/radish
[15]https://tinygo.org/

3 Study Design

Our objective is to help developers manage the cost of their Infrastructure as Code deployments, which we aim to do by building a cost linter for Infrastructure as Code. In this chapter, we lay out our approach towards this goal and towards answering our research questions. A summary of the design of our study is depicted in Figure 3.1. As shown, our study consists of four main parts: data collection, pattern extraction, implementation and evaluation. We detail each step in the following sections. After this study design, we specify a set of requirements that the linter should adhere to.

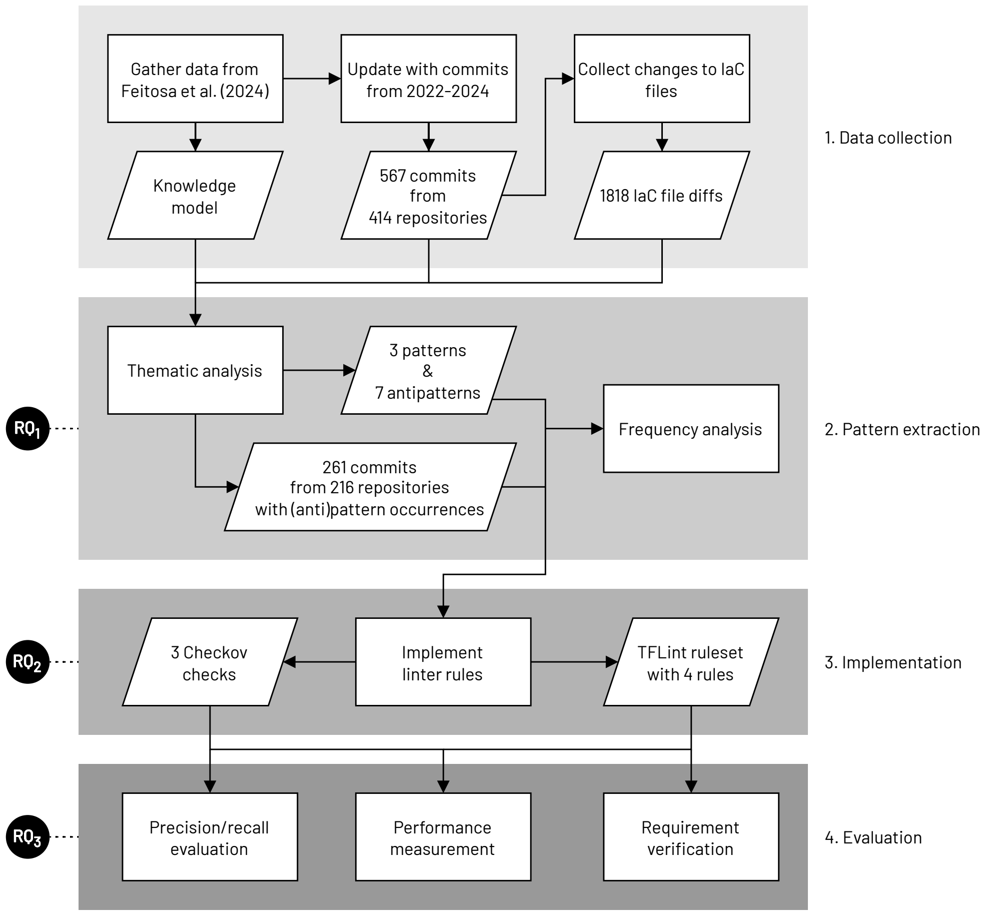

Gather data from Feitosa et al. (2024)
Update with commits from 2022-2024
Collect changes to IaC files
Knowledge model
567 commits from 414 repositories
1818 IaC file diffs
1. Data collection
Thematic analysis
3 patterns & 7 antipatterns
Frequency analysis
261 commits from 216 repositories with (anti)pattern occurrences
RQ1
2. Pattern extraction
RQ2
3 Checkov checks
Implement linter rules
TFLint ruleset with 4 rules
3. Implementation
RQ3
Precision/recall evaluation
Performance measurement
Requirement verification
4. Evaluation

Figure 3.1: Overview of the steps (rectangles) and outputs (parallelograms) of our study design

3.1 Data Collection

For this work, we start from an existing dataset by Feitosa et al. that was constructed in their study on cost awareness in IaC-enabled open-source software [33, 34], specifically projects focusing on Terraform. The choice of Terraform (and its open-source fork, OpenTofu) was motivated by its broad compatibility with cloud service providers, accessible interface, and mature API, which have helped spread adoption in diverse development environments [29]. In addition, GitHub hosts a significantly large volume of Terraform projects, which provided a robust dataset for analysis.

The initial dataset was generated by filtering GitHub repositories that include Terraform descriptor files (`.tf` or `.tf.json`), which are indexed by GitHub. This search targeted repositories created after Terraform's first release in 2014 up to May 2022. An initial pool of 152 735 repositories containing Terraform files was collected for subsequent analysis.

To identify commits indicative of discussions related to cloud cost, a list of keyword stems was employed (`bill`, `cheap`, `cost`, `efficient`, `expens`, and `pay`), designed to capture various expressions of cost-awareness. Next, a set of 6 116 commits (from 2 010 repositories) containing the keywords in their messages was extracted.

From this refined set, a manual review was performed to further ensure the relevance of the commits to cloud cost management. This involved a process where each commit was initially examined by two researchers, with any conflicts resolved in consolidation meetings involving the entire research team. This resulted in a final selection of 538 pertinent commits spanning 434 distinct repositories, which then formed the core data used for further analysis in the prior study.

3.2 Updating the Original Dataset

Because the original study was conducted in May 2022, the dataset only contains commits up until that time. We therefore update the dataset to include commits from June 2022 until May 2024 by re-running the scripts provided by the original study. Then, we perform an initial round of labeling using the approach from the original study with two independent raters. As a measure of agreement, we compute Krippendorff's alpha [47], finding a value of $\alpha = 0.43$. Next, we attempt to resolve conflicts in a resolution meeting, improving the agreement to $\alpha = 0.95$. Finally, a third rater resolves the remaining conflicts in another consolidation meeting. The result is a set of 606 cost-related commits from 445 repositories.

The initial dataset provides the URL, message, and an assigned cost-related label for each commit. For this study, we also need to recover the changes to IaC files, since we focus on them to explore patterns and antipatterns of cost management. For example, although a commit message may address cost of deployment, it is not certain that the code changes reflect the cost-related (part of the) message.

During the commit retrieval we find that 31 repositories are no longer available, resulting in 39 commits not being accessible. From the remaining 567 commits (414 repositories), we extract an average of 4 IaC file diffs per commit (min: 1, median: 2, max: 59). This data collection totals 1 818 file diffs from 1 742 distinct files.

3.3 Pattern Extraction

To answer **RQ$_1$** (what recurring patterns can we find in code changes that address cost issues in the IaC artifacts of cloud software?) and extract meaningful patterns and antipatterns from the commits, we apply thematic analysis [35], a flexible method for qualitative data analysis. This method is particularly suited to our study because it allows for both inductive reasoning, emerging from the data, and deductive reasoning, driven by existing theory and the previous study. Also, this approach has been applied with success in other domains to analyze commits [23, 64]. Inspired by these studies, our thematic analysis process encompasses four stages:

1. **Familiarization with the data:** We first perform a detailed examination of each commit's content, including messages and associated code changes. In addition, the previous study [33] assigned cost-related labels to each commit based on its message, which we also consider.

2. **Generating initial codes:** Through iterative reading and discussion, we develop a set of initial codes that describes the cost-related commit changes. We note that multiple codes may occur in a single commit. These codes are intended to encapsulate key aspects of cost management practices, including the effect (e.g., increase or reduce cost), action (e.g., add, remove or change) and affected property (e.g., computing or network resource). Each code is documented and defined to maintain consistency across the dataset. We also establish a code to identify when no cost-changing actions are identified.

3. **Validating themes:** Codes are then collated into potential themes that reflect broader (anti)patterns in the data. This step involves grouping and regrouping the codes to identify significant trends and outlier practices. We perform multiple rounds of discussion to refine these themes, ensuring they accurately represent the dataset while considering the theoretical framework developed in our previous work. Also, we filter out themes that do not occur in *at least three different repositories*, similar to Cruz and Abreu [23] and Moura et al. [64].

4. **Defining and naming themes:** Each theme is further refined and ultimately defined as a pattern or antipattern. Themes that effectively represent recurring solutions are termed 'patterns,' while those that signify ineffective practices are termed 'antipatterns.' Each (anti)pattern is documented with a (i) brief introduction, (ii) contextual understanding of the underlying IaC problem, (iii) the solution derived from combining the related changes, authors' experience, and logical arguments, and (iv) an example code solution.

3.4 Implementation

To answer **RQ$_2$** (how can these patterns be implemented in a linter?), we aim to implement the (anti)patterns identified in the previous step into a standalone linter or an extension for an existing linter for IaC, in order to help cloud software developers catch and fix cost problems in their codebases, and to enable researchers to further study the occurrence and evolution of these (anti)patterns in IaC-enabled projects over time.

As a first step, we examine the identified (anti)patterns and the associated commit diffs in order to determine the techniques that are required to detect the different issue types, as discussed in Section 2.5. Moreover, we consider the supported targets, methods and extension mechanisms among the landscape of IaC analyzers, as well as the requirements that we define later in this chapter. The combination of this information allows us to select the appropriate implementation approach, i.e. implementing a linter from scratch or extending an existing linter.

Next, drawing inspiration from prior works that have implemented IaC linters such as Brogi, Di Tomasso and Soldani [16] and Rahman et al. [81], we encode the patterns as rules to detect instances of the patterns from properties of Terraform source code, based on the occurrences found in our dataset. Finally, we implement the rules in the rule engines of two linters using the appropriate programming language. We make these rule plugins available for community use and contributions [1] [2]. More information and usage instructions can be found in Section 5.3 and Section 5.4.

3.5 Evaluation

In order to determine the accuracy and performance of our implementation and answer **RQ$_3$** (how well does the resulting linter perform at detecting cost issues?), we evaluate the implementation on two main criteria:

1. *Relevance*, i.e. precision and recall, in terms of true positive, false positive and false negative matches of (anti)patterns;
2. *Performance*, i.e. how quickly the linter can complete a scan.

Besides these criteria, we discuss to which extent the tools that we extend and the respective extensions themselves differ in regards to a set of requirements, defined in Section 3.6.

3.6 Requirements

We list the requirements for the cost linter below. These requirements are the result of problem analysis, our literature survey, and an open discussion. They are also accompanied by a brief justification. The set of requirements is intended to guide both the selection of potential tools to extend, as well as the implementation of rules within such tools. It is therefore a combination of tool requirements (**FR$_2$**, **FR$_3$**, **FR$_5$**, **FR$_6$**, **FR$_7$**, **FR$_8$**, **FR$_9$**, **NFR$_2$** and **NFR$_3$**), and rule requirements (**FR$_1$**, **FR$_4$** and **NFR$_1$**).

[1] https://github.com/InputUsername/checkov/tree/cost-rules
[2] https://github.com/InputUsername/tflint-ruleset-cost

3.6.1 Functional Requirements

The linter must:

FR$_1$ identify cost issues in IaC code.

The linter should be able to analyze IaC languages specifically, as opposed to other languages (e.g. [54, 74]), and should identify cost-related issues, since that is to be the main contribution of our study.

FR$_2$ support other issue types than cost issues alone.

Ideally, the linter should support other issue types (e.g. code or security smells). This effectively means that the cost smell detection should be integrated into an existing linter which already supports other types of issues, because developers report creating and maintaining linter configuration to be a challenge [96] and so reducing the number of static analysis tools required is worthwile.

FR$_3$ identify issues on a granular level beyond the file level.

Several types of smell detectors and static analysis tools, particularly metric-based and ML-based ones, identify issues on a coarse level such as file level, class level or function level. For example, BSDT (metric-based) by Danphitsanuphan and Suwantada [28] detects smells on class and method level. To be useful for developers, the linter should give a precise indication of where the cost smell occurs.

FR$_4$ suggest mitigation strategies to the identified issues.

Finding issues is one thing; knowing how to fix them is another. Developers tend to use linters to prevent errors or issues [96], and suggesting fixes can help them achieve this. Moreover, it could help make developers more aware of the underlying issues [44, 96].

FR$_5$ feature the ability to disable or enable specific checks.

High configurability might help reduce false positives and allow developers to enable only those rules that their project needs [96]. The ability to disable or enable checks on a per-project or even per-analysis level is therefore beneficial to support.

FR$_6$ support regular expression-based rules.

Of the 31 tools for IaC static analysis we identified, 19 support analysis using rules based on regular expressions. While these are not without limitations when it comes to accurate, end-to-end detection of smells in IaC scripts [83], they are easy to implement and modify, and do not require (large) sets of training data like ML-based or metric-based approaches. Thus, it makes sense for our linter to support rules based on regular expressions.

FR$_7$ support graph-based rules.

A minority, 5 out of the 31 tools, support graph analysis techniques or graph-based rules. Graph-based rules may potentially be able to support more complex cost smell detection approaches, making this a useful requirement to have. However, in order not to exclude the majority of tools that only support simpler types of rules, this is not a hard requirement.

FR_8 support IaC languages beyond Terraform.

For our work, we use the dataset collected by Feitosa et al. [34], which only contains data on Terraform artifacts. As such, support for at least Terraform is a hard requirement. However, we expect that some of the identified cost smells might extend to other orchestrators and languages (e.g. CloudFormation). This means that the ability to support additional IaC technologies besides Terraform is useful as well.

FR_9 support integration with integrated development environments (IDEs), at a minimum Visual Studio Code and IntelliJ IDEA.

According to Guerriero et al. [42] who interviewed 44 senior software developers, IDEs are among the most important support tools for IaC development. Their research also specifically mentions VS Code and IntelliJ as prominent examples of IDEs used for IaC. Moreover, an analysis of Google Trends [98] suggests that VS Code and IntelliJ are popular IDEs, with both showing growth in recent years. Together, VS Code and IntelliJ-based IDEs make up 4 out of the top 10 IDEs based on Google Trends activity. It is reasonable to conclude that integration with VS Code and IntelliJ is helpful for the adoption of a (cost) linter.

3.6.2 Non-functional Requirements

The linter must:

NFR_1 raise as few false positives as possible.

Existing literature on static analysis indicates that developers frequently experience false positives [9, 53, 105]. Tomasdottir et al. [96] also found that developers see false positives as a problem, but they do not experience false positives frequently while using ESLint. In their study, Tomasdottir *et al.* speculate that this is due to the highly configurable nature of ESLint as well as the simpler types of analyses provided by linters compared to general static analysis tools. More research is needed to determine whether this extends to IaC analyzers, specifically when looking for cost smells. Nevertheless, minimizing false positives makes sense to prevent developer confusion and frustration.

NFR_2 be reactive and have a short response time, < 500 milliseconds.

This requirement ties in to **FR_9**; developers should receive timely feedback in their editor if cost smells are detected in their IaC code, thus a responsive linter is necessary.

NFR_3 be easy to adopt by developers.

In short, we aim to produce a result that is usable in practice. We expect that supporting multiple issue types (**FR_2**), configurability (**FR_5**), support for many languages (**FR_8**) and IDE integrations (**FR_9**), and limiting false positives (**NFR_1**), will all help developers adopt the tool in their development workflows.

4 Pattern Extraction

In this chapter, we present the results of our thematic analysis process. We note that the contents of this chapter have been adapted as a paper which was accepted at the *50th Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA) 2024* [13]. An accompanying dataset is available online [1].

During the analysis, we identified 161 codes, one denoting "no related changes identified" and the remaining 160 representing various cost-saving or -increasing actions on cloud resources. We found that 368 of the 567 commits contained at least one cost-related code, while 199 commits were coded with "no related changes identified". The top 10 codes (besides "no related changes") are shown in Figure 4.1. The full set of codes and descriptions is available in the online dataset.

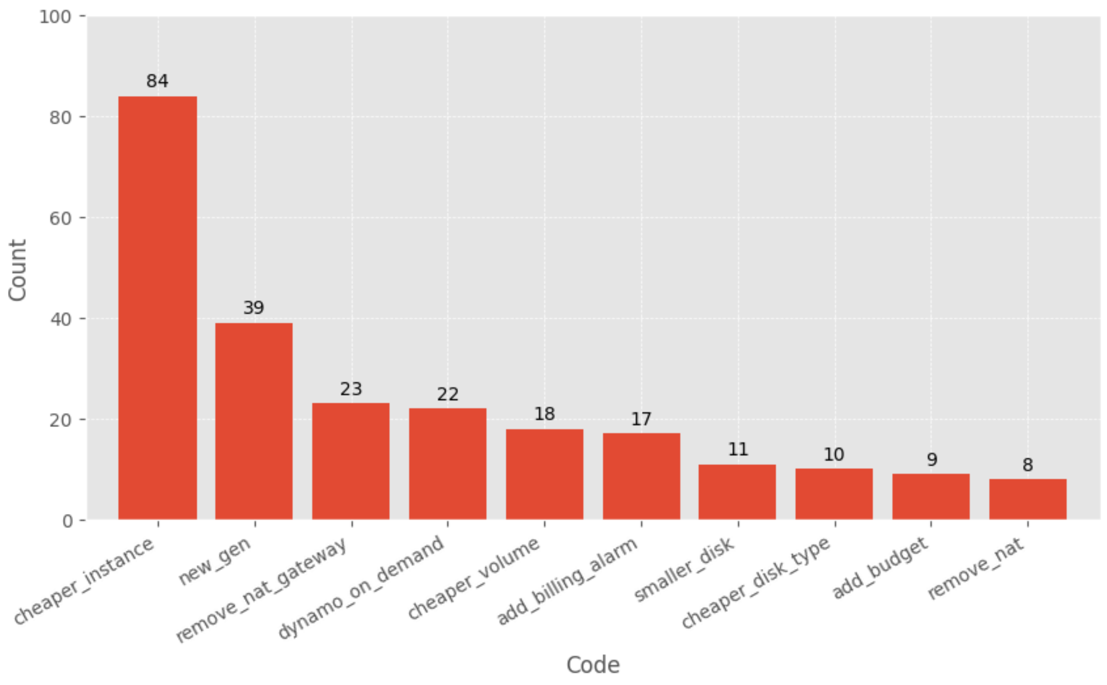


Figure 4.1: Histogram of the top 10 codes before defining the patterns and antipatterns

[1] https://search-rug.github.io/iac-cost-patterns/

After extracting themes using thematic analysis and validating them as discussed in Section 3.3, we identified 3 patterns and 7 antipatterns. We found that 60 of the 161 codes integrated one of the (anti)patterns, while 101 were discarded for lack of relevance (e.g., in a theme present in fewer than three repositories). Ultimately, from the 368 commits coded with at least one cost-related code, 261 contain at least one (anti)pattern. A more detailed frequency analysis of (co-)occurrences is provided in Section 4.2.

4.1 Patterns and Antipatterns

We now list the collected patterns and antipatterns. Links to selected occurrences are available in Table 4.1, and the full list can be found in Appendix A. The occurrences are also included in the online version of the dataset, linked earlier in this chapter.

Table 4.1: Selected occurrences of (anti)patterns

Pattern	**Occurrences**
Budget	AJarombek/global-aws-infrastructure (4a89f4b) MartinFeineis/terraform (359ba42) stuartellis/stuartellis-org-tf-modules (39a9cab)
Spot instances	openinfrastructure/terraform-google-gitlab-runner (8429375) kathputli/terraform-aws (321b1ae) naciriii/terraform-ec2-gitlab-runner (f8af6bc)
Object storage lifecycle rules	alphagov/govuk-aws (f844cd8) alphagov/govuk-terraform-provisioning (ac105ab) ExpediaGroup/apiary-data-lake (47e62f2)
Expensive instance	beaulabs/terraform_aws_ec2_instance (d6df68d) gudlyf/TerraformOpenVPN (4bc861c) IncredibleHolg/infra-aws-code (7090470)
Old generation	gudlyf/TerraformOpenVPN (be1245d) alphagov/govuk-aws (6cfda6a) greenbrian/musical-spork (24c07bf)
Expensive storage type	thomastodon/jabujabu (02210a3) giantswarm/giantnetes-terraform (53ed24b) Kalmalyzer/UE-Jenkins-BuildSystem (ee8942b)
Expensive network resource	stealthHat/k8s-terraform (681a3f8) thomastodon/jabujabu (02210a3) structurefall/jamulus-builder (7190744)
Overprovisioned resources	thomastodon/jabujabu (02210a3) guilhermerenew/infra-cost (ba858d9) chaspy/terraform-alibaba-isucon8 (53588da)
AWS - Expensive DynamoDB	deptno/terraform-aws-modules (49f447b) ONSdigital/eq-terraform-dynamodb (40eb651) olliefr/aws-terraform-cloud1 (bf75383)
Expensive monitoring	Eximchain/terraform-aws-quorum-cluster (6a56f40) Accurate0/infrastructure (06889e0) cloudspout/Gefjun (665692a)

Pattern - Budget

Use budgets to receive alerts about charged and forecast costs and control spending.

Context: The lack of explicit cost monitoring can often lead to unforeseen and undesirable costs.

Solution: Major cloud providers support the creation of budgets, which allow users to define alerts about charged and forecast costs and control spending. Having one or more budgets can help monitor and manage the cost of cloud deployments.

Example: Define a budget for a cost limit of 1200 USD for EC2, and generate an email notification if the forecast monthly cost exceeds this amount:

```
resource "aws_budgets_budget" "example" {
  name              = "example"
  budget_type       = "COST"
  limit_amount      = "1200"
  limit_unit        = "USD"
  time_unit         = "MONTHLY"

  cost_filter {
    name = "Service"
    values = [
      "Amazon Elastic Compute Cloud - Compute",
    ]
  }

  notification {
    comparison_operator        = "GREATER_THAN"
    threshold                  = 100
    threshold_type             = "PERCENTAGE"
    notification_type          = "FORECASTED"
    subscriber_email_addresses = ["test@example.com"]
  }
}
```

Pattern - Object storage lifecycle rules

Define lifecycle rules for object storage to move objects to cheaper storage or drop them entirely.

Context: By default, objects stored in cloud object storage are retained, and therefore billed, indefinitely. Objects also have a storage class or access tier, which can be used to balance access performance and cost depending on the use case.

Solution: By configuring lifecycle rules or policies, objects can be transitioned to cheaper storage classes or deleted after a certain amount of time.

Example: Transition objects under the ”log/” prefix to the Glacier storage class after 60 days, and expire after 90 days:

```
resource "aws_s3_bucket_lifecycle_configuration" "example" {
  bucket = aws_s3_bucket.bucket.id

  rule {
    id = "log"

    expiration {
      days = 90
    }

    filter {
      prefix = "log/"
    }

    status = "Enabled"

    transition {
      days          = 60
      storage_class = "GLACIER"
    }
  }
}
```

Pattern - Spot instances

Use spot instances to run interruptible workloads for significant cost savings compared to regular instances.

Context: Continuously running compute instances are also continuously billed. Certain types of workloads which can handle interruption, e.g. batch jobs, data analysis and optional tasks, do not require on-demand, provisioned instances.

Solution: Major cloud providers offer excess compute capacity in the form of spot instances. These provide discounts over on-demand compute instances, with the caveat that instances can be preempted or deleted at any time when compute capacity needs to be reclaimed. Users define a price limit and if the spot price falls below this limit, an instance is allocated. If a user's workloads can handle interruptions, spot instances can offer an economical alternative to regular instances.

Example: Use spot instances to run batch jobs: if some of the instances are preempted, the job is slowed down, but it does not completely stop. For example, request a worker at a price of 0.03 USD:

```
resource "aws_spot_instance_request" "cheap_worker" {
  # ...
  spot_price    = "0.03"
  instance_type = "c4.xlarge"

  tags = {
    Name = "Worker"
  }
}
```

Antipattern - Expensive instance

Compute instances are often overprovisioned even when a cheaper instance would suffice.

Context: A recurring pattern in cloud deployments is that developers initially choose compute instances which are overprovisioned, because it is difficult to know the requirements upfront. This leads to situations where developers deploy, for example on AWS, '2xlarge' instances, when in fact 'large' or even 'medium' would suffice.

Solution: Critically evaluate required performance levels and special functionality (e.g. memory-optimized versus general-purpose instances), and scale down the provisioned instance types where appropriate.

Example: Downgrade to a cheaper general-purpose instance in the same family to save costs:

```
@@ -1,5 +1,5 @@
 resource "google_compute_instance" "example" {
   name         = "example"
-  machine_type = "n1-standard-1"
+  machine_type = "g1-small"
   # ...
 }
```

Antipattern - Old generation

Using newer resource generations gives similar performance for lower cost.

Context: Cloud providers occasionally update their offerings to support, for example, newer CPU generations. These newer generations are often more efficient, making them a more economical option compared to older generations.

Solution: Upgrade resources to newer generations to attain comparable or better performance for a lower price. The most commonly replaced resources include, but are not limited to, AWS's t2 general-purpose compute instances and gp2 storage volumes.

Example: Switch from gp2 to gp3 storage, providing comparable performance but lower cost:

```
@@ -1,7 +1,7 @@
 resource "aws_instance" "example" {
   # ...
   root_block_device {
-    volume_type = "gp2"
+    volume_type = "gp3"
     # ...
   }
 }
```

Antipattern - Expensive storage type

More expensive storage types are often used even when cheaper storage types would be sufficient.

Context: Developers are able to choose between different storage types (HDD vs SSD, durability guarantees) for e.g. instances' root disks. However, not all use cases require highly durable SSD storage, making cheaper storage types a viable way to save cost.

Solution: Evaluate performance and durability guarantees for storage and switch to a less expensive type where relevant.

Example: Switch an OS disk from Premium LRS storage to Standard LRS:

```
@@ -1,6 +1,6 @@
 resource "azurerm_linux_virtual_machine" "example" {
   # ...
   os_disk {
-    storage_account_type = "Premium_LRS"
+    storage_account_type = "Standard_LRS"
   }
 }
```

Antipattern - Expensive network resource

Network resources like NAT gateways, elastic IP addresses and subnets tend to be expensive while not being strictly needed.

Context: Due to their interdependence, the cost of certain types of networking resources often adds up. For example, a developer may create multiple subnets, each having its own NAT gateway, each of which in turn is assigned an IPv4 address. In other cases, network resources are used which are not strictly required, e.g. load balancers.

Solution: It is often possible to forgo the use of the expensive resources entirely. Solutions include subnets sharing a single NAT gateway, reducing the number of subnets or removing the use of load balancers.

Example: Remove resources that are not strictly required, or reduce the number of networking resources. For example, the commonly used module terraform-aws-modules/vpc has an option to use a single NAT gateway instead of creating one per subnet:

```
module "vpc" {
  source = "terraform-aws-modules/vpc"

  # ...

  enable_nat_gateway = true
  single_nat_gateway = true
}
```

Antipattern - Overprovisioned resources

Resources like RAM, storage and CPU utilization are often overprovisioned even when lower values are acceptable.

Context: In a similar way to overprovisioned instances, it is difficult to estimate required limits for resources such as root storage upfront, leading developers to overprovision them, in turn raising costs.

Solution: Evaluate the resource requirements and lower the relevant values.

Example: Shrink the root storage size of an instance to reduce storage costs:

```
@@ -2,6 +2,6 @@ resource "aws_instance" "example" {
   root_block_device {
-    volume_size = 20 # GB
+    volume_size = 15 # GB
   }
 }
```

Antipattern - AWS - Expensive DynamoDB

AWS DynamoDB tables often use features that carry cost but are not required, especially for infrequently accessed tables.

Context: DynamoDB tables might use provisioned billing mode, have high (> 1) read/write capacity, or use global secondary indices. These features carry additional cost and are not always required, especially for infrequently accessed tables.

Solution: Switching to pay-per-request billing mode, reducing provisioned read/write capacity, and removing global secondary indices are ways to cost-optimize DynamoDB tables.

Example: Set billing mode to pay-per-request:

```
resource "aws_dynamodb_table" "example_table" {
  name         = "HighScores"
  billing_mode = "PAY_PER_REQUEST"

  attribute {
    name = "UserID"
    type = "S"
  }

  attribute {
    name = "Score"
    type = "N"
  }
}
```

Antipattern - Expensive monitoring

Monitoring solutions are expensive and might not be needed.

Context: Cloud providers offer ways to monitor deployed infrastructure and collect metrics and logs. These solutions add cost for e.g. health checks and log storage, and the benefits may not outweigh this cost.

Solution: Removing monitoring or logs for noncritical infrastructure is an effective way to save cost.

Example: Remove a Route 53 health check for a private Plex instance to save costs:

```
@@ -1,6 +0,0 @@
-resource "aws_route53_health_check" "example" {
-  fqdn = "plex.example.com"
-  port = 443
-  request_interval = "30"
-  failure_threshold = "5"
-}
```

4.2 (Co-)occurrences

The occurrences of (anti)patterns and their co-occurrences within the same commit or within the same repository are summarized by the UpSet plots of Figure 4.2 and Figure 4.3, respectively. As it can be seen in the figures, the most frequent ones are the *Expensive instance* and *Expensive network resources* antipatterns. Not surprisingly, these two antipatterns are also the two most frequent ones overall. The most frequent pattern, on the other hand, is *Budget* with 27 commits across 27 distinct repositories. This could be indicative of projects independently having their "moment of illumination" that specific cost items need to be kept under control and imposing budget limits accordingly. The rest of the patterns are definitely less frequently occurring in comparison to that.

What is more interesting, however, is that the same antipatterns tend to occur repeatedly in different commits from the same repositories. The most extreme example of this is the *Old generation* antipattern occurring in 6 different commits of the (now deprecated) AWS Terraform repository [2] for `gov.uk` applications from 5 different dates between 2019 and 2021. This recurrence points towards persistent problems with bringing the same infrastructural aspects under control over time that needs further investigation for its root cause.

Looking specifically at co-occurrences, from the long tail in both figures it can easily be observed that two or more (anti)patterns co-occur relatively rarely, even when looking at the granularity of repositories. Some of these co-occurrences are expected, e.g. *Expensive instance* and *Old generation*, but some of them point to more complex, structural problems, e.g. *Expensive instance* and *Overprovisioned resources*. What Figure 4.3 cannot show is the ordering in which they occurred and the actions to address them. Investigating this is beyond the scope of our work.

[2] https://github.com/alphagov/govuk-aws

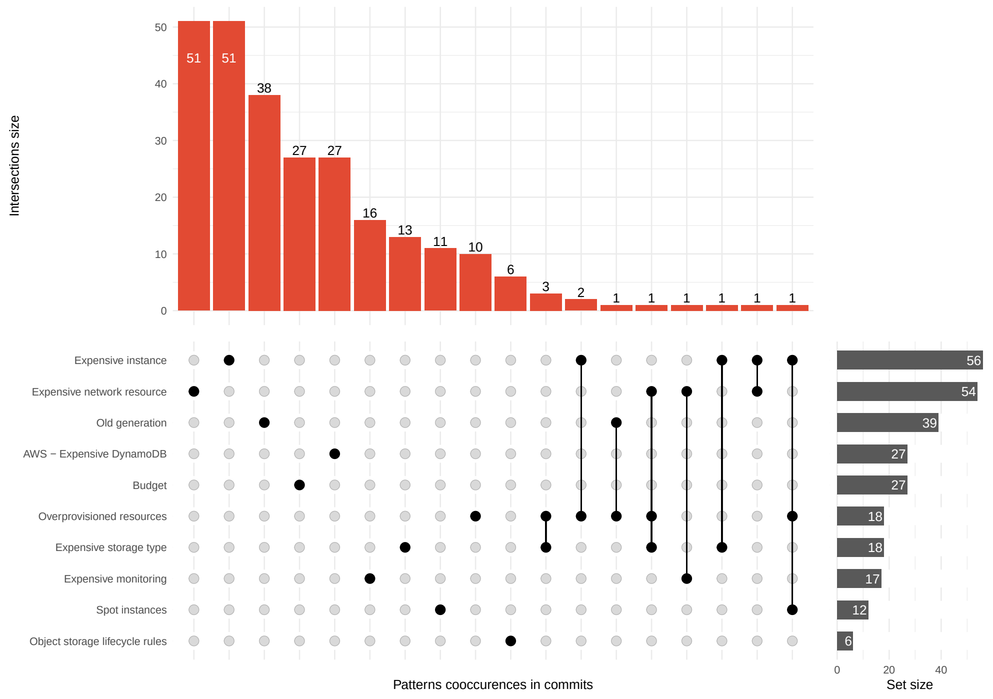


Figure 4.2: (Co-)occurrences of (anti)patterns within commits, adapted from Bolhuis, Feitosa and Andrikopoulos [13]

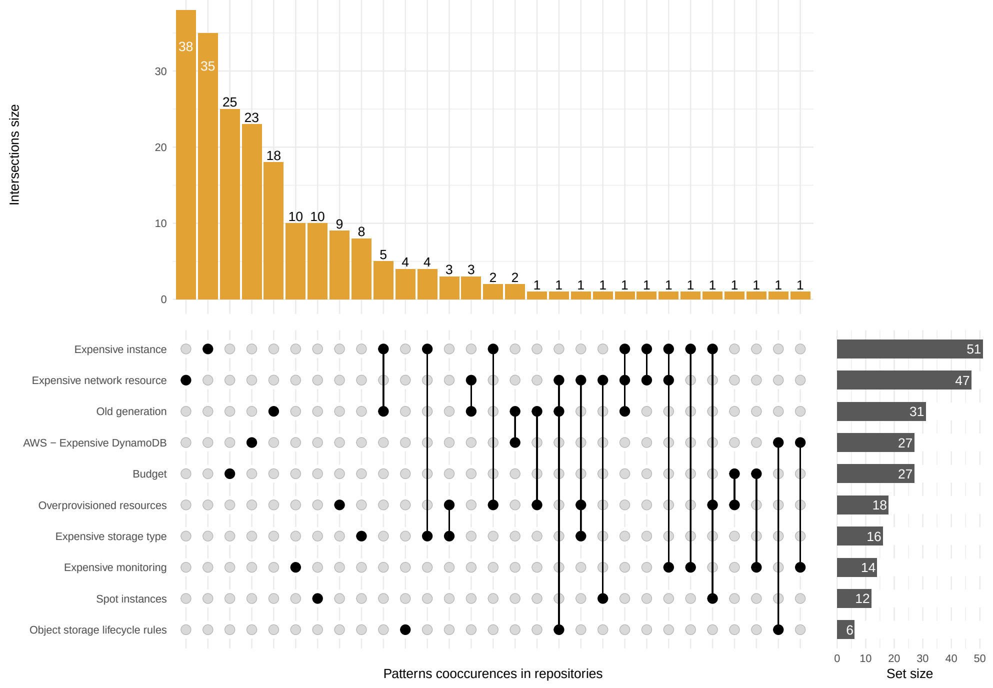


Figure 4.3: (Co-)occurrences of (anti)patterns within repositories, adapted from Bolhuis, Feitosa and Andrikopoulos [13]

5 Implementation

Our hope is that the pattern catalog introduced in the previous chapter is a useful contribution for practitioners in and of itself. However, considering the many advantages of static analysis tools in general and linters in particular, we would like to go a step further and detect these patterns in an automated fashion. In this chapter, we discuss the translation of the patterns into linter rules and the implementation thereof.

5.1 Tool Selection

From the requirements for language support (**FR$_8$**) and IDE support (**FR$_9$**), as well as the breadth of already available IaC analysis tools, we can conclude that consolidating efforts in an existing tool is preferable to creating a tool from scratch. We start from the IaC analyzers in Table 2.1; other tools referenced in Chapter 2 are already excluded by **FR$_8$**, as we found none that support Terraform, and adding Terraform support would be beyond the scope of this project. We then follow the steps in Table 5.1 to select a target for our extension based on the requirements and the functionality offered by the respective tools.

After this filtering process, the tools that remain are Checkov [18] and TFLint [95]. Both tools are popular, open-source, industry-backed IaC linters, with 6 753 and 4 756 GitHub stars, respectively, as of July 2024. Because we cannot further separate them based on their functionality and our requirements, we implement the linter rules in both Checkov and TFLint and compare the results in our evaluation. However, before we proceed with the implementation, we briefly discuss each tool.

Checkov is a linter with support for many different IaC technologies and formats, written in Python. It supports Terraform's HCL and JSON syntaxes and plan files, AWS CloudFormation JSON and YAML templates and Azure Resource Manager templates, among others. It is aimed at detecting common misconfigurations and comes with an extensive set of over 1 200 rules, which it calls policies or checks. Checkov's primary focus is on detecting security smells and deviations from best practices, but in principle it can detect arbitrary types of issues. It implements its own parsers to convert input files into an intermediate format. In the case of Terraform, this involves parsing resource definitions and building a graph of resource connections, after which individual resource configurations are passed to checks for inspection. Two check types are supported: *attribute checks*, which inspect (combinations of) attributes to determine whether a resource "passes" or "fails" the check, and *connection checks*, which pass or fail depending on whether a resource is connected to some other resource of a given type. Besides

Table 5.1: Tool exclusion process

	Excluded Tools	Reason
1	ACID [80], GASEL [67], Häyhä [59], Rehearsal [91], Opdebeeck *et al.* [68]	No source code available.
2	Snyk IaC Security [50]	Free tier limited to 300 tests per month [1].
3	terrafirma [104], tfsec [6]	Deprecated.
4	Ansible Lint [4], BARREL [15], Bicep linter [100], cookstyle [19], DeepIaC [14], foodcritic [38], GLITCH [86], Puppeteer [92], RADON [71], SLAC [81], SLIC [77], SODALITE [58], Sommelier [16], TAMA [46]	No Terraform support (**FR$_8$**).
5	terrascan [94], terraform-compliance [70], Regula [84]	No IDE integrations (**FR$_9$**).
6	Semgrep [90], trivy [7]	No variable evaluation, no cross-file analyses [2] [3].
7	KICS [56]	Requires multiple positive and negative examples per query, increasing implementation effort.
8	SonarLint	No documented extension mechanism.

its built-in policies, Checkov also features an extension mechanism, using which developers can add custom checks. User-defined checks can be written in Python or a YAML-based domain-specific language, though connection checks are only supported in the latter.

TFLint, on the other hand, is a Terraform-oriented linter. It is written in Go, and supports inspections on Terraform files that use the HCL format. TFLint does not focus on any particular category of issues and does not come with any rules built-in; instead, it is built using a pluggable architecture, with inspection rules provided as plugins. These so-called *rulesets* are executed by the host process and communicate bi-directionally over gRPC [4]: TFLint sends inspection requests to the ruleset plugins, after which the plugins can request information about the Terraform configurations to be analyzed. This architecture is further detailed in Appendix B. TFLint implements its own HCL parser, which was forked from Terraform's codebase. As a result, it supports arbitrary inspections, and rulesets can even implement checks for syntax errors. Several official rulesets are available for the "big three" cloud providers (AWS, Azure and GCP), as well as a plugin for general Terraform language errors and best practices. Each plugin is written in Go using the TFLint plugin SDK. Plugins implement one or more rules,

[1] https://snyk.io/plans/
[2] https://semgrep.dev/pricing
[3] https://aquasecurity.github.io/trivy/v0.52/docs/advanced/modules/
[4] https://grpc.io/

which request Terraform configurations, inspect them, and emit zero or more issues as a result. Rules define metadata such as a documentation or reference URL and a severity (notice, warning or error), while each issue contains a message and a source range specifying the lines and columns where the issue occurred.

5.2 Mapping Patterns to Rules

Similar to past work [16, 77, 81], we define rules for detecting instances of (anti)patterns in our pattern catalog. We select two patterns and two antipatterns: *Budget*, *Object storage lifecycle rules*, *Old generation* and *AWS - Expensive DynamoDB*. We select these because they constitute (anti)patterns that can apply unconditionally. By contrast, the remaining patterns and antipatterns are conditional on the infrastructural requirements of a system. For example, it is nontrivial to determine up-front how much root storage a server will require or how much memory a Lambda needs, which makes it difficult to detect the *Overprovisioned resources* antipattern.

Our catalog contains both patterns and antipatterns. However, most, if not all linters use rules that detect the presence of some error, issue or bad practice. We therefore reframe the meaning of patterns such that a rule for a pattern triggers if the pattern is not applied, thus treating the *absence of a pattern* as an antipattern.

We extract the rules by examining the diffs of the commits in which each (anti)pattern occurs, as well as the codes associated with the commit. In this way, we can formulate expressions that succinctly describe the conditions that must hold for the rule to apply.

Although the patterns and antipatterns in our catalog are, for the most part, provider-agnostic, our linter rules are not. Due to the fact that the overwhelming majority of commits—over 72%, as highlighted in Appendix C—were made to systems that use AWS, only three occurrences are systems that use another provider.

In the rule definitions, we use the following notation:

- type(r) denotes the resource type of r, e.g. 'aws_instance';
- r.name refers to an *argument* or *attribute* of r [5], e.g. r.bucket;
- Resources can contain *blocks* [6], and predicates of the form has(...)Block(r) indicate the existence of such a block in resource r.

It is also important to note that while existing studies' rules do not consider features like variable evaluation, ours do, because Terraform implements that functionality, and support is included in both Checkov and TFLint. The rules are thus assumed to be applied to the *expanded* configuration, with e.g. variables substituted and loops evaluated.

[5] https://developer.hashicorp.com/terraform/language/syntax/configuration#arguments

[6] https://developer.hashicorp.com/terraform/language/syntax/configuration#blocks

5.2.1 Budget

$$\begin{aligned}&\neg\exists r : (\text{type}(r) = \text{‘aws\_budgets\_budget’}\\&\quad\vee (\text{type}(r) = \text{‘aws\_cloudwatch\_metric\_alarm’} \wedge r.\text{metric\_name} = \text{‘EstimatedCharges’})\\&\quad\vee \text{type}(r) = \text{‘google\_budgets\_budget’})\end{aligned}$$

The *Budget* pattern states that it is a good practice to define budgets for infrastructure that can warn about excessive forecast costs. Thus, while other rules apply to specific resource definitions, the rule for *Budget* triggers if there is no budget or metric alarm for estimated charges.

5.2.2 Object storage lifecycle rules

$$\begin{aligned}&\text{type}(r_1) = \text{‘aws\_s3\_bucket’} \wedge \neg\text{hasLifecycleBlock}(r_1)\\&\wedge \neg\exists r_2 : (\text{type}(r_2) = \text{‘aws\_s3\_bucket\_lifecycle\_configuration’} \wedge r_2.\text{bucket} = r_1.\text{id})\end{aligned}$$

Following the *Object storage lifecycle rules*, we recommend developers define lifecycle rules for object storage solutions in order to delete data which is no longer required or transition it to cheaper storage tiers. Terraform's AWS provider supports two ways of defining these rules for an `aws_s3_bucket`: (1) using one or more `lifecycle_rule` blocks; (2) using a `aws_s3_bucket_lifecycle_configuration` pointing at the bucket. Lack of either therefore triggers the rule for a given S3 bucket r_1.

5.2.3 Old generation

The *Old generation* antipattern applies to two main resource types: `aws_instance` and `aws_ebs_volume`. For clarity, we define a separate rule for each resource.

Instances

$$\text{type}(r) = \text{‘aws\_instance’} \wedge \text{isOldInstanceType}(r.\text{instance\_type})$$

In our analysis, we found that developers most often tend to move away, for cost reasons, from the `t2` and `m4` instance classes. We therefore define the predicate isOldInstanceType(t) to be true when the instance type t contains the regular expression pattern `'t2|m4'`.

Volumes

$$\begin{aligned}&\text{type}(r) = \text{‘aws\_instance’} \wedge \text{hasRootVolumeBlock}(r)\\&\wedge \text{isOldVolumeType}(r.\text{root\_volume.volume\_type})\\&\vee \text{type}(r) = \text{‘aws\_ebs\_volume’} \wedge \text{isOldVolumeType}(r.\text{type})\end{aligned}$$

Volumes can be defined in multiple ways, including as the root volume of an instance or as a standalone Elastic Block Storage (EBS) volume. Similar to instances, isOldVolumeType(t) is defined to be true if the volume type matches `'gp2'`, which developers universally move away from for cost reasons.

5.2.4 AWS - Expensive DynamoDB

$$\begin{aligned}
&\text{type}(r) = \text{`aws\_dynamodb\_table'} \land (r.\text{billing\_mode} \neq \text{`PAY\_PER\_REQUEST'} \\
&\lor r.\text{read\_capacity} > 1 \lor r.\text{write\_capacity} > 1 \\
&\lor \text{hasGlobalSecondaryIndexBlock}(r))
\end{aligned}$$

With *AWS - Expensive DynamoDB*, we identified three DynamoDB table configurations to be cost-ineffective:

- Not using pay-per-request billing mode, i.e. using provisioned mode;
- Using read and/or write capacities higher than one;
- Defining global secondary indices.

If any of these conditions hold, it indicates an expensive DynamoDB table.

5.3 Rule Implementation

As a proof-of-concept, we have implemented the aforementioned detection rules in Checkov and TFLint, as listed in Table 5.2. These implementations are available on GitHub: a fork of Checkov with the custom checks [7], and a ruleset plugin for TFLint [8]. Usage instructions are presented in Section 5.4.

Table 5.2: (Anti)patterns implemented as linter rules

(Anti)pattern	**Checkov**	**TFLint**
Budget		✓
Object storage lifecycle rules	✓	✓
Old generation	✓	✓
AWS - Expensive DynamoDB	✓	✓

5.3.1 Checkov

Apart from *Budget*, all (anti)patterns are implemented for Checkov. As discussed before, Checkov's rule engine applies its rules to individual resources. This means it cannot perform arbitrary inspections on Terraform configurations, like detecting the absence of a budget [9]. Support for such rules could in principle be added by extending the rule engine, but doing so is outside the scope of this project.

Regular checks are implemented in Python by extending the `BaseResourceCheck` class. Custom checks can be scaffolded using an interactive prompt in Checkov's command-line interface, which also sets up the correct file structure for the check to work:

[7] https://github.com/InputUsername/checkov/tree/cost-rules
[8] https://github.com/InputUsername/tflint-ruleset-cost
[9] https://github.com/bridgecrewio/checkov/issues/4926

```
checkov --add-check
```

An example of the structure of a check is provided in Listing 5.1. Checks define a description, unique identifier, one or more supported resources, a guideline URL and a set of categories, primarily security-oriented. Then, the `scan_resource_conf` method defines how to validate a resource. The configuration is passed in as a Python dictionary, which is then used to check (part of) the rule for *AWS - Expensive DynamoDB*.

```python
class DynamoDbPayPerRequest(BaseResourceCheck):
    def __init__(self):
        # This is the full description of your check
        description = "Ensure that DynamoDB tables use PAY_PER_REQUEST billing mode"

        # This is the Unique ID for your check
        id = "CKV_AWS_801"

        # These are the terraform objects supported by this check (ex:
        ↪  aws_iam_policy_document)
        supported_resources = ['aws_dynamodb_table']

        guideline = 'https://search-rug.github.io/(...)'

        # Valid CheckCategories are defined in checkov/common/models/enums.py
        categories = [CheckCategories.CONVENTION]
        super().__init__(name=description, id=id, categories=categories,
        ↪  supported_resources=supported_resources, guideline=guideline)

    def scan_resource_conf(self, conf):
        if 'billing_mode' not in conf.keys() or 'PAY_PER_REQUEST' not in
        ↪  conf['billing_mode']:
            self.details.append('Using provisioned billing mode might incur
            ↪  unnecessary cost for infrequently accessed tables')
            return CheckResult.FAILED

        return CheckResult.PASSED


check = DynamoDbPayPerRequest()
```

Listing 5.1: Checkov check written in Python, which checks the billing mode of a DynamoDB table

YAML checks need to be created manually. They use a domain-specific language [10], which is based on combining attribute checks and connection checks with boolean operators to define the conditions which cause the check to pass. Checks can furthermore use operators such as `exists`, `contains` and `regex_match` to validate resource attributes and connections to other resources.

Listing 5.2 shows an example of a YAML check. In fact, it is a check that is already included in Checkov [11], which fails when an S3 bucket does not have a connected lifecycle configuration or does not define a lifecycle rule block. However, it is not clear whether

[10] https://www.checkov.io/3.Custom%20Policies/YAML%20Custom%20Policies.html

[11] https://github.com/bridgecrewio/checkov/blob/main/checkov/terraform/checks/graph_checks/aws/S3BucketLifecycle.yaml

the check was created with cost considerations in mind and does not come with a message informing the developer of any cost concerns. Nevertheless, we use the existing rule since it fully covers the *Object storage lifecycle rules* pattern and maps one-to-one to our rule definition.

```yaml
metadata:
  name: "Ensure that an S3 bucket has a lifecycle configuration"
  category: "LOGGING"
  id: "CKV2_AWS_61"
definition:
  or:
    - and:
        - cond_type: filter
          attribute: resource_type
          operator: within
          value:
            - aws_s3_bucket
        - cond_type: connection
          resource_types:
            - aws_s3_bucket
          connected_resource_types:
            - aws_s3_bucket_lifecycle_configuration
          operator: exists
    - cond_type: attribute
      resource_types:
        - aws_s3_bucket
      attribute: lifecycle_rule
      operator: exists
```

Listing 5.2: Checkov check written in YAML, which ensures an S3 bucket defines a lifecycle configuration

5.3.2 TFLint

As discussed earlier, TFLint is extended using a plugin system, and plugins are standalone programs written in Go that use the TFLint plugin SDK [12] [13] to communicate with the host program. The SDK exposes functions to enumerate resource definitions, which also take an optional schema that can be used to select which attributes and blocks will be required for inspection. In addition, there are functions to evaluate variables, and functions to emit issues with or without fixes.

TFLint's approach is shown in more detail in Listing 5.3, a simplified version of the rule for *AWS - Expensive DynamoDB*. Rules are defined as structs on which several methods are implemented. The main function to be implemented is `Check`, which takes a handle to a runner, i.e. a connection to the main process, and implements the actual inspection by querying resources of type `"aws_dynamodb_table"`, including any `"global_secondary_index"` blocks, and emitting an issue for each block.

A caveat that should be mentioned is that TFLint can only partially implement *Object storage lifecycle rules*; TFLint cannot evaluate cross-resource references [14], so the con-

[12] https://github.com/terraform-linters/tflint-plugin-sdk
[13] https://pkg.go.dev/github.com/terraform-linters/tflint-plugin-sdk
[14] https://github.com/terraform-linters/tflint/blob/v0.51.2/docs/user-guide/compatibility.md#unsupported-named-values

```go
func (r *CostAwsExpensiveDynamoDbRule) Check(runner tflint.Runner) error {
  tables, err := runner.GetResourceContent("aws_dynamodb_table", &hclext.BodySchema{
    Attributes: []hclext.AttributeSchema{},
    Blocks: []hclext.BlockSchema{
      {Type: "global_secondary_index", Body: &hclext.BodySchema{}},
    },
  }, nil)
  if err != nil {
    return err
  }

  for _, table := range tables.Blocks {
    for _, globalSecondaryIndex := range table.Body.Blocks {
      if err := runner.EmitIssue(r, "global secondary indices are expensive",
      ↪  globalSecondaryIndex.DefRange); err != nil {
        return err
      }
    }
  }

  return nil
}
```

Listing 5.3: TFLint rule, which detects the use of global secondary indices on DynamoDB tables

nection between `aws_s3_bucket` and `aws_s3_bucket_lifecycle_configuration` cannot be checked, only the presence of a `lifecycle_rule` block.

5.4 Usage

To use the set of cost checks with Checkov, they can be extracted [15], placed in a directory, and pointed at using the `--external-checks-dir` command-line argument. Alternatively, the fork can be built and installed locally [16]. The checks have been developed and tested against Checkov version `3.2.109`. To run Checkov, the following command invocation can be used:

```
checkov --evaluate-variables true \
    --download-external-modules true \
    --framework terraform \
    --check <check id 1,check id 2,...> \
    --directory "/path/to/project"
```

This runs the specified (comma-separated) checks on the target directory and subdirectories, evaluating variables, downloading external modules and checking local and external modules, while filtering all files except `.tf` and `.tf.json` files.

To install the TFLint ruleset, `tflint-ruleset-cost` can be downloaded and installed by executing `make install` from the root directory. This will build the ruleset and

[15] https://github.com/InputUsername/checkov/tree/cost-rules/checkov/terraform/checks/resource/aws/cost/

[16] https://github.com/bridgecrewio/checkov/blob/main/CONTRIBUTING.md#build-package-locally

copy it to the correct directory. Alternatively, it can be distributed as a GitHub release and installed by following the steps in the TFLint developer guide [17]. The ruleset was developed and tested with TFLint version `0.51.1`. TFLint can be executed as follows:

```
cd /path/to/project
terraform get
tflint --call-module-type=all \
    --recursive \
    --enable-plugin=cost \
    --only="<rule name 1>" --only="<rule name 2>"
```

This command uses the specified ruleset plugin, enabling only specific rules, and performs an inspection while calling both local and remote external modules. Note that to evaluate external modules, they first need to be downloaded using `terraform get`. Furthermore, TFLint accepts a parameter `--chdir=path`, but this will only inspect the project root directory. Instead, the `--recursive` flag can be used to inspect all directories, though it is mutually exclusive with `--chdir`, which means changing the working directory with `cd` beforehand is required.

Examples of performing a scan with Checkov and TFLint can be found in Appendix D.

[17] https://github.com/terraform-linters/tflint/blob/master/docs/developer-guide/plugins.md#4-creating-a-github-release

6 Evaluation

Having discussed the implementation of a set of linter rules in the previous chapter, we now proceed with an evaluation of this implementation. We quantify the relevance of the results returned by both linters, conduct a performance evaluation, and determine the fulfillment of the requirements.

6.1 Relevance

In line with prior work [81, 86], we evaluate the precision and recall of the implemented linter rules. Precision relates to the share of true positive and false positive matches compared to all matches. Recall on the other hand relates to the share of true positives and false negatives, i.e. the ability not to miss existing issues.

6.1.1 Setup and Results

Whereas the existing studies compute the precision and recall based on sets of files which have been manually classified by smell type, we do not have that luxury: our dataset only consists of a set of commits categorized by the type of issue they are addressing. We also do not have access to labeled files or commits beyond our dataset because of the effort required in labeling. We further decide to evaluate only using commits from our dataset because of the relatively small number of commits which can be scanned successfully, as a result of e.g. parsing errors, or errors accessing external modules. For improved confidence in the results, a more comprehensive evaluation should be performed in future work. The total number of commits used to evaluate Checkov is 72, while TFLint also has the occurrences associated with *Budget*, leading to a total of 99 commits.

While we know which (IaC) files are involved in each commit, we argue that this set of files is too limited due to Terraform's use of external modules, variable evaluation and cross-file references. This means that running Checkov and TFLint against those files that the commit modifies is not enough.

Instead, we create a snapshot of the parent commit(s) of each commit in our dataset [1] to obtain a repository state where the addressed (anti)pattern is present. We report the

[1] Two commits did not have a parent, i.e. were initial commits, and were therefore filtered out:

- `chetanbothra/Terraform_AWS_Billing_Alert` (hash: `43b0d3b`)
- `openaustralia/infrastructure` (hash: `63ee190`)

precision and recall in terms of issues detected or missed by Checkov and TFLint in Table 6.1.

For completeness, we also snapshot the repository state after each commit, since this gives us a set of repositories where the addressed (anti)pattern is no longer present and should thus not trigger the linter rules. The precision and recall for this extended set of repository states can be found in Table 6.2.

Table 6.1: Precision and recall of Checkov and TFLint ("before" state)

	Checkov			**TFLint**		
(Anti)pattern	***Count***	***Precision***	***Recall***	***Count***	***Precision***	***Recall***
Budget	-	-	-	15	0.32	0.47
Object storage lifecycle rules	3	0.11	1.00	2	0.14	1.00
Old generation	31	0.80	0.52	10	0.67	0.60
AWS - Expensive DynamoDB	23	0.82	1.00	14	1.00	0.93

Table 6.2: Precision and recall of Checkov and TFLint ("before" and "after" states)

	Checkov			**TFLint**		
(Anti)pattern	*Count*	***Precision***	*Recall*	*Count*	***Precision***	*Recall*
Budget	-	-	-	15	0.15	0.47
Object storage lifecycle rules	3	0.06	1.00	2	0.08	1.00
Old generation	31	0.48	0.56	10	0.40	0.60
AWS - Expensive DynamoDB	23	0.53	1.00	14	0.81	0.93

As shown, overall results are mixed. In the "before" state, Checkov achieves a precision ≥ 0.8 for *Old generation* and *AWS - Expensive DynamoDB*, while its precision is low for *Object storage lifecycle rules* because of a large number of false positives. Checkov misses about half the issues for *Old generation*, but achieves 1.0 recall for the other (anti)patterns. TFLint meanwhile has poor precision for *Budget* and *Object storage lifecycle rules*, also as a result of a large number of false positives, but it performs better for *Old generation* and *AWS - Expensive DynamoDB*. TFLint's recall is ≥ 0.9 for *Object storage lifecycle rules* and *AWS - Expensive DynamoDB*, while just under 50% and 60% of occurrences of *Budget* and *Old generation*, respectively, are detected correctly.

By introducing the "after" state, naturally precision will remain equal at best, because in this state, issues should theoretically be addressed. This also means that recall should not change, assuming a commit that fixes one issue does not introduce another. From the table, we can see that indeed recall stays the same, but precision drops across the board following an increase in false positives.

Differences in occurrence counts between Checkov and TFLint might be explained by the differences in parsing, variable evaluation and resolution of external modules between the two tools; we expect that TFLint's use of Terraform's own parsing code leads to stricter enforcement of language rules and thus more repositories causing errors.

6.1.2 False Positives

The computed precision values could be deceptive because valid matches of (anti)patterns, which are not fixed by a commit, are counted as false positives: like we determined in Section 4.2, most commits only address one (anti)pattern at a time, while multiple linter rules can (correctly) trigger for one commit. In fact, for Checkov, 23 commits triggered two or more rules ("before" state; 13 in the "after" state), and for TFLint, 27 commits triggered multiple rules (26 "after"). To illustrate this, we have collected a number of examples of such wrongly labelled false positives.

The project `trajano/terraform-s3-backend` addresses the *AWS - Expensive DynamoDB* antipattern (commit hash: `f4b61c7`). However, upon manual inspection, its "before" state (hash: `905fb70`) does not define any budget or billing alarm, causing the rule for *Budget* to correctly trigger in TFLint.

Another example is `circleci/enterprise-setup` (now archived), whose commit (hash: `26cc529`) addresses the *Old generation* antipattern, but the "before" state (hash: `f8c42ed`) also triggers the rule for *Object storage lifecycle rules* in both Checkov and TFLint. The offending AWS S3 bucket [2], as seen in Listing 6.1, indeed does not define any lifecycle rules and is not connected to a lifecycle configuration.

```
resource "aws_s3_bucket" "circleci_bucket" {
  # VPC ID is used here to make bucket name globally unique(ish) while
  # uuid/ignore_changes have some lingering issues
  bucket = "${replace(var.prefix, "_", "-")}-bucket-${replace(var.aws_vpc_id, "vpc-",
↪  "")}"

  cors_rule {
    allowed_methods = ["GET"]
    allowed_origins = ["*"]
    max_age_seconds = 3600
  }

  force_destroy = var.force_destroy_s3_bucket
}
```

Listing 6.1: Incorrect false positive for *Object storage lifecycle rules*

The project `olliefr/aws-terraform-cloud1`, which addresses *AWS - Expensive DynamoDB* in its commit (hash: `bf75383`), also triggers the rules for Old generation in Checkov and TFLint in its "before" state (hash: `d0464cc`). Inspecting the AWS EC2 instance [3], shown in Listing 6.2, confirms that this is a valid match: the instance type is `"t2.micro"`.

[2] https://github.com/CircleCI-Archived/enterprise-setup/blob/f8c42ed15fc935c477a213ebdd69ac55af14e932/circleci.tf#L59-L71

[3] https://github.com/olliefr/aws-terraform-cloud1/blob/d0464cce1d4fe314e9c15354b6567eaea409bbfe/example.tf#L27-L30

```
resource "aws_instance" "example" {
  ami           = "ami-04edc9c2bfcf9a772"
  instance_type = "t2.micro"
}
```

Listing 6.2: Incorrect false positive for *Old generation*

While TFLint achieves 1.00 precision for *AWS - Expensive DynamoDB*, Checkov finds several instances in repositories that have gone unaddressed. One example is the (archived) project `dwp/dataworks-aws-data-egress`, which addresses *Old generation* in a commit (hash: `14f065e`) but has an expensive DynamoDB configuration [4] that can be seen in Listing 6.3, by not using pay-per-request billing mode and using high provisioned read/write capacity.

```
resource "aws_dynamodb_table" "data_egress" {
  name           = "data-egress"
  hash_key       = "source_prefix"
  range_key      = "pipeline_name"
  read_capacity  = 20
  write_capacity = 20

  attribute {
    name = "source_prefix"
    type = "S"
  }

  attribute {
    name = "pipeline_name"
    type = "S"
  }

  tags = merge(
    local.common_tags,
    {
      Name = "data-egress"
    },
  )
}
```

Listing 6.3: Incorrect false positive for *AWS - Expensive DynamoDB*

6.1.3 Latest Commits

The previous subsection gives some credibility to the existence of instances of (anti)patterns in repositories that have gone unaddressed. In Table 6.3, we list the number of occurrences in the repositories' latest commits. We additionally filter for repositories that are active, that is, have been updated within the last 6 months. Although further analysis is beyond the scope of this thesis, and we cannot with confidence rule out false positives, these cases might hint at persistent problems both within and across projects.

[4] `https://github.com/dwp/dataworks-aws-data-egress/blob/e9b3269f53ac9c0a6cca7ec4932ee50d1b99a148/data-egress.tf#L12-L35`

Table 6.3: Occurrences of patterns in (active) repositories' latest commits

	Checkov		**TFLint**	
Pattern	***All***	***Active***	***All***	***Active***
Budget	-	-	31	10
Object storage lifecycle rules	23	10	18	6
Old generation	6	2	6	3
AWS - Expensive DynamoDB	11	2	5	0

6.2 Performance

To determine performance, we measure the scan duration of Checkov and TFLint during evaluation. Figure 6.1 shows the average inspection duration per repository for both the "before" and "after" states. As shown, Checkov takes around 30 seconds for a full repository scan, while TFLint takes slightly less than half a second. Figure 6.2, with outliers removed, reveals that both tools have similar distributions, with relatively low median scan duration but a longer tail on the upper end.

Results are dominated by a major outlier, though: the top four commits that took longest to analyze with both Checkov (mean: 377.0 seconds) and TFLint (mean: 5.6 seconds) all belong to the `ministryofjustice/cloud-platform-environments` project, a large, actively-maintained repository containing over 7 700 Terraform files. This relation between repository size (in terms of the number of lines of IaC code) and scan duration is shown in more detail in Figure 6.3; the four outlier commits can clearly be distinguished.

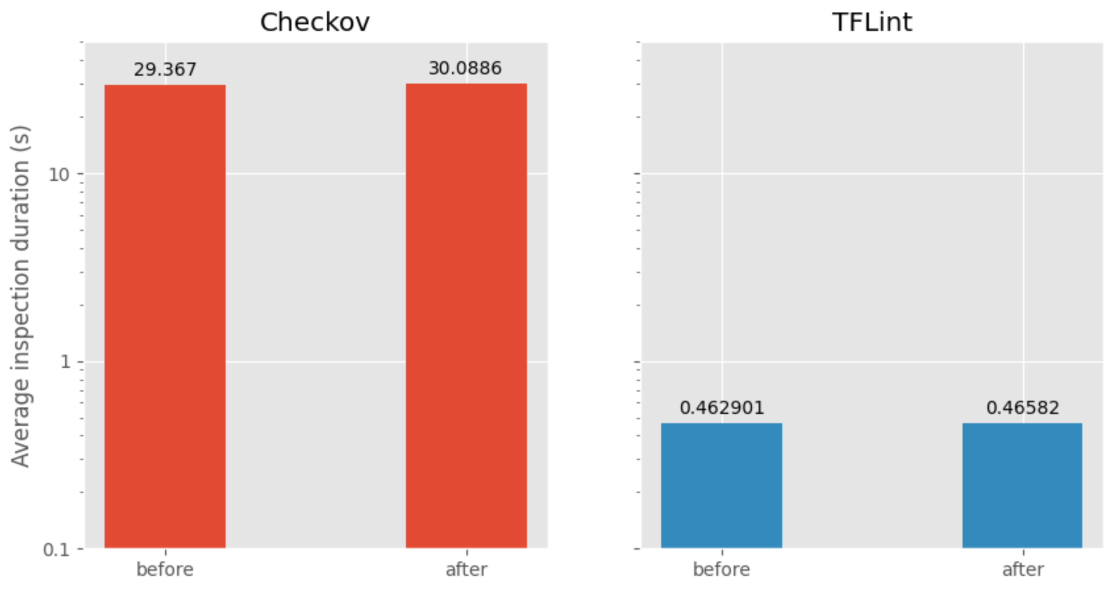


Figure 6.1: Average inspection duration for Checkov and TFLint

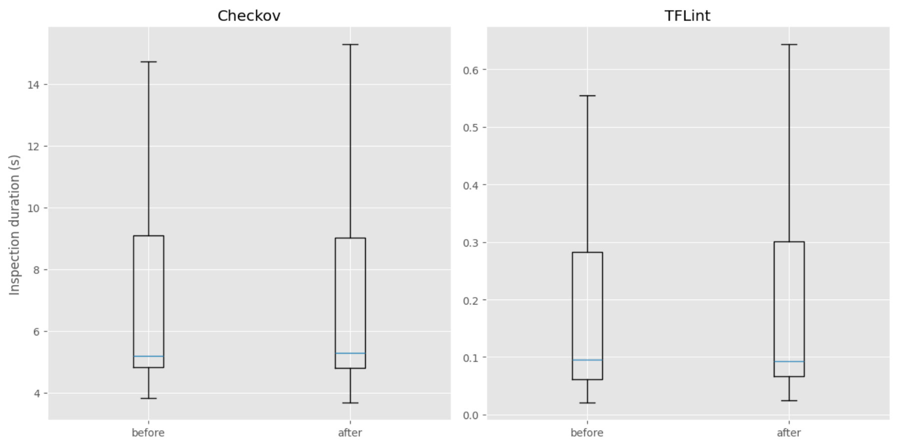


Figure 6.2: Comparison of inspection duration for Checkov and TFLint

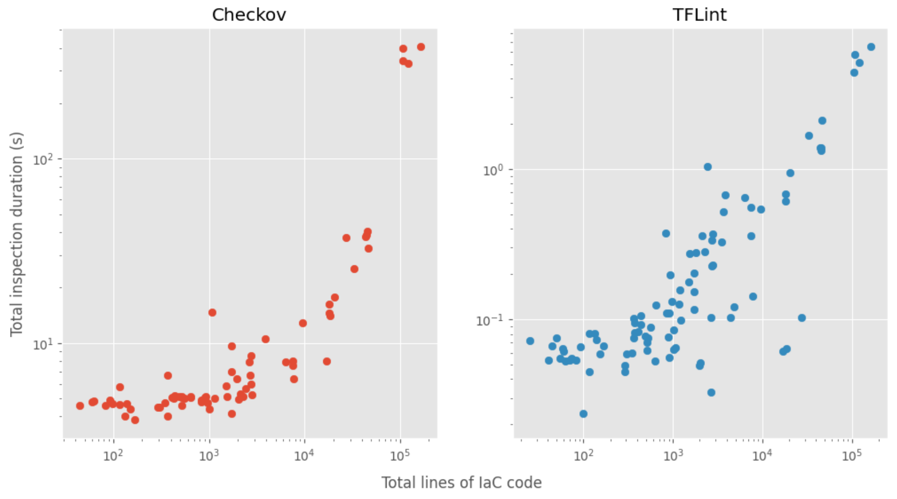


Figure 6.3: Relation between the number of lines of IaC code and inspection duration

6.3 Comparison Between Checkov and TFLint

Overall, both Checkov and TFLint are able to support cost (anti)pattern detection, in addition to other issue types like security and code smells. Both tools also offer key linter functionality, including the ability to precisely point out the location of an issue—Checkov on the level of resources or attributes, and TFLint on the level of line and column spans—as well as the ability to supply documentation-related information along with emitted issues, and simple regular expression rules. There are (minor) differences in precision and recall, but in general, neither implementation is free from false positives or false negatives. Looking at detection capabilities, Checkov can support resource connection checks, whereas TFLint supports rules to detect resource existence. An advantage of choosing Checkov is the flexibility afforded by parsing artifacts to an intermediary format, enabling support for more IaC languages than Terraform. This is contrasted by its performance, however, with TFLint scans being over an order of magnitude faster on average.

In Table 6.4, we list to what extent each implementation fulfills the requirements specified in Section 3.6. We distinguish three levels of fulfillment: *fulfilled* (indicated by ●), *partially fulfilled* (◐) and *not fulfilled* (○).

Table 6.4: Fulfillment of requirements

		Fulfillment	
Requirement	**Description**	*Checkov*	*TFLint*
$\mathbf{FR_1}$	Identify cost smells	●	●
$\mathbf{FR_2}$	Support issues beyond cost smells	●	●
$\mathbf{FR_3}$	Granularity beyond file level	●	●
$\mathbf{FR_4}$	Suggest mitigation strategies	●	●
$\mathbf{FR_5}$	Ability to disable or enable specific checks	●	●
$\mathbf{FR_6}$	Support regular expression rules	●	●
$\mathbf{FR_7}$	Support graph rules	●	○
$\mathbf{FR_8}$	Language support beyond Terraform	●	○
$\mathbf{FR_9}$	Support IDE integration	◐	◐
$\mathbf{NFR_1}$	Limit false positives	◐	◐
$\mathbf{NFR_2}$	Short response time	○	●
$\mathbf{NFR_3}$	Easy to adopt by developers	◐	◐

As is shown in the table, most functional requirements are fulfilled by both the Checkov and TFLint implementations, with a number of exceptions. As discussed in Section 5.3.2, TFLint does not support graph rules ($\mathbf{FR_7}$). It also only supports Terraform and so it does not satisfy $\mathbf{FR_8}$. The requirement for IDE integration, $\mathbf{FR_9}$, is partially achieved: while both tools support IDE integration, Checkov's plugins have been deprecated in favor of the (paid) Prisma Cloud [5] service and its associated plugins. Meanwhile, TFLint can be executed in Language Server Protocol (LSP) mode, but does not have IDE plugins itself. It does however come packaged as part of MegaLinter [6], which has plugins for Visual Studio Code and IntelliJ IDEA.

Following Section 6.1, Checkov and TFLint partially fulfill $\mathbf{NFR_1}$. The precision for certain (anti)patterns is quite high (≥ 0.8), while for others it is not. $\mathbf{NFR_2}$ is fully fulfilled by TFLint, achieving an average response time for a full-project scan of 0.46 seconds, but not by Checkov, which took around 30 seconds per project on average. Finally, since it is effectively a combination of $\mathbf{FR_2}$, $\mathbf{FR_5}$, $\mathbf{FR_8}$, $\mathbf{FR_9}$ and $\mathbf{NFR_1}$, it follows that $\mathbf{NFR_3}$ is also partially fulfilled by both tools.

[5] https://www.prismacloud.io/
[6] https://megalinter.io/latest/descriptors/terraform_tflint/

7 Discussion

In this chapter, we discuss the findings for each phase of our study to answer the research questions, as well as some of the limitations in this work.

7.1 Pattern Extraction

First, to find recurring themes in the way developers address cost issues in IaC artifacts (**RQ$_1$**), we carefully analyzed a set of 567 commits on Terraform files from 414 open-source repositories, and through doing so curated a catalog of 3 patterns and 7 antipatterns specifically relating to IaC cost management. The predominance of antipatterns may suggest that developers are primarily in a reactive state, addressing issues only after they are introduced, as opposed to proactively avoiding them. Recognizing these antipatterns, like using excessively provisioned resources or outdated resource classes, is essential because they pinpoint where costs could escalate without proper management. On the other hand, the identified patterns provide effective strategies to improve cost efficiency, showcasing steps developers can take to prevent problems.

The analysis of (co-)occurrences reveals ongoing cost-related issues in projects. The repeated occurrence of the same antipatterns in various commits of a single repository indicates a need for a more systematic cost management approach. This repeated struggle with similar problems highlights the importance of having a structured framework or toolkit to help developers consistently apply best practices and avoid common mistakes.

7.2 Implementation

Our literature search and list of IaC static analysis tools showed that existing research and industry-supported tools are primarily concerned with security and code smells. Nevertheless, we found two suitable IaC linters, Checkov and TFLint, which we were able to extend with detection rules for cost issues (**RQ$_2$**). We did so by analyzing the commit diffs associated with two patterns and two antipatterns, and specifying a set of detection rules based on the Terraform constructs involved with each issue. Then, we translated these rules into checks for Checkov written in Python, and TFLint rules written in Go, and we made these implementations available online. In this way, we have provided an initial step towards automated detection of cost issues in Infrastructure as Code.

Both tools do show shortcomings in terms of the types of rules that can be implemented, though, with e.g. Checkov's lack of resource existence checks and TFLint's limitation in scanning cross-resource dependencies. Besides implementation details, it is also worth considering whether it even makes sense to detect certain (anti)patterns. For example, some types of resources may be more expensive but required for enhanced network isolation, such as virtual private clouds. In those cases a tradeoff needs to be made between cost and other factors, which is difficult to do in an automated fashion.

7.3 Evaluation

The evaluation of our implementation (**RQ_3**) showed mixed results in terms of precision and recall. Part of the low precision can be explained by the limitations of our evaluation method and the fact that we did not have the means to label instances of patterns that were never fixed. We suspect that the true number of false positives is lower than the precision suggests. The cases of low recall for the *Budget* and *Old generation* (anti)patterns are likely due to limitations in our implementation. For *Budget*, there were several cases where developers moved away from AWS CloudWatch billing alarms in favor of dedicated budgets, but since the detection rule checks for the existence of either of those resources, the "absence of budget" is not detected in the before state. For *Old generation*, we selected a subset of resource classes (`t2` and `m4` instances, `gp2` volumes) that were most common, but of course this means that cases like old instance types for AWS Relational Database Service (RDS) servers and other old generation resources are not detected.

In spite of the low precision and recall for certain (anti)patterns, we were able to identify cases where false positives were actually correct matches. This further strengthens the idea that developers would benefit from a systematic approach to cost management as opposed to repeatedly fixing one-off issues. What's more, we also found that occurrences of patterns and antipatterns exist even in repositories' latest commits, a surprising fact given that all repositories are involved in at least one cost-related commit, showing cost awareness among the contributors of these projects. Although we have not been able to verify how many false positives are among these matches, this is an interesting result that may warrant future research.

In regards to performance, there is a stark difference between Checkov and TFLint. Where TFLint fully complies with **NFR_2** (response time under 500 milliseconds), Checkov is more than an order of magnitude slower. The difference in implementation language (Python versus Go) likely plays a large part in this, as well as the architectures of the respective programs. We expect that long scan durations can be partially mitigated by incremental scans and caching (commonly implemented by IDE plugins), but evaluating this is outside the scope of this project.

All in all, most requirements are at least partially achieved, and we certainly expect either implementation to have utility in developers' workflows. Which of Checkov or TFLint is better cannot be concluded with certainty; both tools have their strengths and weaknesses when it comes to detection capabilities, responsiveness and other features like IDE integration.

7.4 Threats to Validity

Being an empirical work, there a threats to its validity, which we discuss in the following. The threats are based on the guidelines presented by Wohlin et al. [106].

Construct validity refers to the connection between what we intend to measure and what we ultimately end up measuring. Our work relies on a rigorous thematic analysis of the identified commits. Although robust, this method may miss instances where the rationale is implicit rather than explicitly mentioned in the commit message.

We defined themes based on occurrences in at least 3 different repositories, in order to ensure that they represent recurring practices as opposed to isolated cases. While this approach produces more generalizable patterns, it might overlook less common but potentially impactful practices.

Regarding the evaluation of the detection accuracy of our implementation, we are limited by the nature of our dataset. Due to the additional effort required to label unaddressed cost issues, we cannot confirm which false positive results are in fact unlabeled but valid issues. This means the reported numbers for precision may not be fully representative.

External validity involves threats to the generalizability of our results. The fact that our work focuses solely on open-source repositories from GitHub that use Terraform as their IaC tool may limit the generalizability of our pattern catalog and the derived linter rules. While another thesis has shown the (anti)patterns to extend to AWS CloudFormation [65], and while the catalog also applies to Terraform's fork OpenTofu, future research could improve the representativeness of the catalog and linter rules by analyzing repositories that use other cloud orchestration tools. This is especially relevant given the large number of labels in our dataset with only one or two occurrences, where extending the dataset may reveal new patterns and antipatterns.

Whereas the patterns and antipatterns that we defined are not provider-specific, their occurrences in commits are. Over 72% of commits were found in projects that use Amazon Web Services as their cloud provider, compared to 12% and 10% for Google Cloud and Azure (respectively), which resulted in the extracted detection rules essentially being AWS-specific. Many of AWS's (Terraform) resources do have analogous versions in other providers, which means this bias towards AWS could be resolved in the future.

Reliability concerns the bias introduced by the researchers involved in data collection or analysis. Manual filtering and thematic analysis have the potential to introduce subjective biases. As a mitigation step, our analysis takes into account the codes assigned by Feitosa et al. [33], which were carefully and iteratively defined and refined by multiple researchers. Moreover, our own coding process also underwent scrutiny involving multiple researchers to ensure consistency and objectivity.

Finally, to mitigate other possible threats to the reliability of our work, we documented the analysis process and provided a dataset containing our documented list of codes, indicators and (anti)patterns. The collection of the initial set of commits is also thoroughly documented and replicable [33, 34].

8 Conclusion

With this work, our goal was to help developers of Infrastructure as Code-enabled cloud software better manage the costs of their deployments by creating an automated tool that can detect potential cost issues.

As a first step, we analyzed a set of commits and their diffs to classify the cost-saving or -increasing actions taken by developers, producing a dataset of commits labeled with these actions. From this, we extracted patterns that developers can apply to their IaC manifests to manage cost, as well as antipatterns that should be avoided.

This pattern catalog can already be a useful educational resource for developers, but automatically catching issues during development would be even more helpful. We therefore also transferred several (anti)patterns to two popular IaC linters, Checkov and TFLint. Our hope is that this can help developers proactively manage their cloud spend as early in the development cycle as possible. Despite limitations in Checkov and TFLint, imperfect detection and the presence of false positives, we were able to find unaddressed instances of patterns and antipatterns in open-source repositories, suggesting that either tool might help developers find common cost issues in their IaC codebases.

8.1 Future Work

Given the lack of prior art in regards to IaC-specific cost management, and taking into account some of the limitations of our study, there are many different avenues for future research:

- **Implementing more (anti)patterns as linter rules:** we have implemented a number of patterns as a proof-of-concept, but an obvious step would be to implement more (anti)patterns as rules to make the linter(s) more comprehensive.
- **Extending the work to other cloud orchestrators:** our work limits itself to Terraform because the dataset we use only focuses on Terraform files. However as mentioned earlier, another thesis has shown that patterns and antipatterns from our catalog also occur in projects using CloudFormation [65], and we expect some of the (anti)patterns to extend to other orchestrators as well. This would mean an increase in coverage of the catalog, and a possibility of implementing rules for a linter targeting those technologies.
- **Determining the prevalence and evolution of (anti)patterns:** in Section 6.1.2, we established that multiple repositories contain instances of the cost (anti)patterns

that, to our knowledge, have never been addressed, despite the developers' clear awareness of certain cost issues. Therefore, by running the implemented linter rules against open-source repositories, it may be possible to identify how often issues occur "in the wild" and when—or if—they are addressed throughout projects' lifetimes.

- **Reaching out to developers:** building on the previous point, finding out through developer outreach if developers are making cost-related changes which are not explicitly identified as such, and why developers are or are not addressing certain issues.
- **Investigating the effectiveness of ML models such as large language models (LLMs):** in line with the majority of IaC static analysis tools [83], our implementation uses regular expressions and ad-hoc rules to detect cost issues. These offer flexibility and simplicity at the cost of accuracy. As we discussed in Chapter 2, ML-based detection approaches come with drawbacks that make them challenging to use in a linter. However, it may be worth investigating if and how models like LLMs could be used to detect cost smells in IaC artifacts, e.g. by using cloud providers' dedicated cost optimization guides in their reasoning process.

References

[1] Francisco Alfredo, André L. Santos, and Nuno Garrido. “Sprinter: A Didactic Linter for Structured Programming”. In: *OpenAccess Series in Informatics* 102 (July 2022), pp. 1–2. ISSN: 21906807. DOI: 10.4230/OASICS.ICPEC.2022.2/-/STATS.

[2] Nabil Almashfi and Lunjin Lu. “Code smell detection tool for java script programs”. In: *2020 5th International Conference on Computer and Communication Systems, ICCCS 2020* (May 2020), pp. 172–176. DOI: 10.1109/ICCCS49078.2020.9118465.

[3] Vasilios Andrikopoulos et al. “How to adapt applications for the Cloud environment: Challenges and solutions in migrating applications to the Cloud”. In: *Computing* 95.6 (June 2013), pp. 493–535. ISSN: 0010485X. DOI: 10.1007/S00607-012-0248-2/TABLES/3.

[4] *ansible/ansible-lint: ansible-lint checks playbooks for practices and behavior that could potentially be improved and can fix some of the most common ones for you*. URL: https://github.com/ansible/ansible-lint.

[5] *aquasecurity/cloudsploit: Cloud Security Posture Management (CSPM)*. URL: https://github.com/aquasecurity/cloudsploit.

[6] *aquasecurity/tfsec: Security scanner for your Terraform code*. URL: https://github.com/aquasecurity/tfsec.

[7] *aquasecurity/trivy: Find vulnerabilities, misconfigurations, secrets, SBOM in containers, Kubernetes, code repositories, clouds and more*. URL: https://github.com/aquasecurity/trivy.

[8] Matej Artac et al. “DevOps: Introducing infrastructure-as-code”. In: *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017* (June 2017), pp. 497–498. DOI: 10.1109/ICSE-C.2017.162.

[9] Al Bessey et al. “A few billion lines of code later”. In: *Communications of the ACM* 53.2 (Feb. 2010), pp. 66–75. ISSN: 00010782. DOI: 10.1145/1646353.1646374.

[10] Betsy Beyer et al. *Site Reliability Engineering: How Google Runs Production Systems*. O’Reilly Media, Inc., 2016. ISBN: 9781491929124.

[11] Farzana Ahamed Bhuiyan and Akond Rahman. “Characterizing Co-located Insecure Coding Patterns in Infrastructure as Code Scripts”. In: *Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASEW 2020* (Sept. 2020), pp. 27–32. DOI: 10.1145/3417113.3422154.

[12] Koen Bolhuis. *Supplementary Material for the Master’s Thesis ”Catching Cost Issues in Infrastructure as Code Artifacts using Linters”*. 2024. DOI: 10.5281/zenodo.13149367.

[13] Koen Bolhuis, Daniel Feitosa, and Vasilios Andrikopoulos. "A Catalog of Cost Patterns and Antipatterns for Infrastructure as Code". In: *2024 50th Euromicro Conference on Software Engineering and Advanced Applications*. 2024.

[14] Nemania Borovits et al. "DeepIaC: Deep learning-based linguistic anti-pattern detection in IaC". In: *MaLTeSQuE 2020 - Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation, Co-located with ESEC/FSE 2020* (Nov. 2020), pp. 7–12. DOI: 10.1145/3416505.3423564.

[15] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. "Modelling and analysing cloud application management". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9306 (2015), pp. 19–33. ISSN: 16113349. DOI: 10.1007/978-3-319-24072-5_2.

[16] Antonio Brogi, Antonio Di Tommaso, and Jacopo Soldani. "Sommelier: a tool for validating TOSCA application topologies". In: *Communications in Computer and Information Science* 880 (2018), pp. 1–22. ISSN: 18650929. DOI: 10.1007/978-3-319-94764-8_1.

[17] William J Brown et al. *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

[18] *checkov*. URL: https://www.checkov.io/.

[19] *chef/cookstyle: A linting tool that helps you to write better Chef Infra cookbooks and In-Spec profiles by detecting and automatically correcting style, syntax, and logic mistakes in your code.* URL: https://github.com/chef/cookstyle.

[20] Wei Chen, Guoquan Wu, and Jun Wei. "An Approach to Identifying Error Patterns for Infrastructure as Code". In: *Proceedings - 29th IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW 2018* (Nov. 2018), pp. 124–129. DOI: 10.1109/ISSREW.2018.00-19.

[21] Michele Chiari, Michele De Pascalis, and Matteo Pradella. "Static Analysis of Infrastructure as Code: A Survey". In: *2022 IEEE 19th International Conference on Software Architecture Companion, ICSA-C 2022* (2022), pp. 218–225. DOI: 10.1109/ICSA-C54293.2022.00049.

[22] Ram Chillarege et al. "Orthogonal Defect Classification—A Concept for In-Process Measurements". In: *IEEE Transactions on Software Engineering* 18.11 (1992), pp. 943–956. ISSN: 00985589. DOI: 10.1109/32.177364.

[23] Luis Cruz and Rui Abreu. "Catalog of energy patterns for mobile applications". In: *Empirical Software Engineering* 24.4 (Aug. 2019), pp. 2209–2235. ISSN: 15737616. DOI: 10.1007/S10664-019-09682-0.

[24] Ting Dai et al. "Automatically detecting risky scripts in infrastructure code". In: *SoCC 2020 - Proceedings of the 2020 ACM Symposium on Cloud Computing* (Oct. 2020), pp. 358–371. DOI: 10.1145/3419111.3421303.

[25] Stefano Dalla Palma, Dario Di Nucci, and Damian A. Tamburri. "AnsibleMetrics: A Python library for measuring Infrastructure-as-Code blueprints in Ansible". In: *SoftwareX* 12 (July 2020), p. 100633. ISSN: 2352-7110. DOI: 10.1016/J.SOFTX.2020.100633.

[26] Stefano Dalla Palma et al. "Toward a catalog of software quality metrics for infrastructure code". In: *Journal of Systems and Software* 170 (Dec. 2020), p. 110726. ISSN: 0164-1212. DOI: 10.1016/J.JSS.2020.110726.

[27] Stefano Dalla Palma sdallapalma et al. "Singling the odd ones out: A novelty detection approach to find defects in infrastructure-as-code". In: *MaLTeSQuE 2020 - Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation, Co-located with ESEC/FSE 2020* 20 (Nov. 2020), pp. 31–36. DOI: 10.1145/3416505.3423563.

[28] Phongphan Danphitsanuphan and Thanitta Suwantada. "Code smell detecting tool and code smell-structure bug relationship". In: *2012 Spring World Congress on Engineering and Technology, SCET 2012 - Proceedings* (2012). DOI: 10.1109/SCET.2012.6342082.

[29] Leonardo Reboucas De Carvalho and Aleteia Patricia Favacho De Araujo. "Performance Comparison of Terraform and Cloudify as Multicloud Orchestrators". In: *Proceedings - 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020* (May 2020), pp. 380–389. DOI: 10.1109/CCGRID49817.2020.00-55.

[30] Seema Dewangan et al. "A novel approach for code smell detection: An empirical study". In: *IEEE Access* 9 (2021), pp. 162869–162883. ISSN: 21693536. DOI: 10.1109/ACCESS.2021.3133810.

[31] Dario Di Nucci et al. "Detecting code smells using machine learning techniques: Are we there yet?" In: *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings* 2018-March (Apr. 2018), pp. 612–621. DOI: 10.1109/SANER.2018.8330266.

[32] Amin Milani Fard and Ali Mesbah. "JSNOSE: Detecting javascript code smells". In: *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013* (2013), pp. 116–125. DOI: 10.1109/SCAM.2013.6648192.

[33] Daniel Feitosa et al. "Mining for cost awareness in the infrastructure as code artifacts of cloud-based applications: An exploratory study". In: *Journal of Systems and Software* 215 (Sept. 2024), p. 112112. ISSN: 0164-1212. DOI: 10.1016/J.JSS.2024.112112.

[34] Daniel Feitosa et al. *Supplementary Material for Mining Cost Awareness in the Infrastructure as Code Artifacts of Cloud-based Applications*. 2024. DOI: 10.5281/ZENODO.11319775.

[35] Jennifer Fereday and Eimear Muir-Cochrane. "Demonstrating Rigor Using Thematic Analysis: A Hybrid Approach of Inductive and Deductive Coding and Theme Development". In: *International Journal of Qualitative Methods* 5.1 (Mar. 2006), pp. 80–92. ISSN: 1609-4069. DOI: 10.1177/160940690600500107.

[36] Francesca Arcelli Fontana et al. "Automatic metric thresholds derivation for code smell detection". In: *International Workshop on Emerging Trends in Software Metrics, WETSoM* 2015-August (Aug. 2015), pp. 44–53. ISSN: 23270969. DOI: 10.1109/WETSOM.2015.14.

[37] Francesca Arcelli Fontana et al. "Code smell detection: Towards a machine learning-based approach". In: *IEEE International Conference on Software Maintenance, ICSM* (2013), pp. 396–399. DOI: 10.1109/ICSM.2013.56.

[38] *Foodcritic/foodcritic: Lint tool for Chef cookbooks.* URL: https://github.com/Foodcritic/foodcritic.

[39] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[40] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN: 0201633612.

[41] Olivier Le Goaer. "Enforcing Green Code with Android Lint". In: *Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASEW 2020* (Sept. 2020), pp. 85–90. DOI: 10.1145/3417113.3422188.

[42] Michele Guerriero et al. "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry". In: *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019* (Sept. 2019), pp. 580–589. DOI: 10.1109/ICSME.2019.00092.

[43] Di Guo and Haitao Wu. "A Review of Bad Smells in Cloud-based Applications and Microservices". In: *Proceedings - 2021 International Conference on Intelligent Computing, Automation and Systems, ICICAS 2021* (2021), pp. 255–259. DOI: 10.1109/ICICAS53977.2021.00059.

[44] Sarra Habchi, Xavier Blanc, and Romain Rouvoy. "On adopting linters to deal with performance concerns in android apps". In: *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* 11 (Sept. 2018), pp. 6–16. DOI: 10.1145/3238147.3238197.

[45] Mohammed Mehedi Hasan, Farzana Ahamed Bhuiyan, and Akond Rahman. "Testing practices for infrastructure as code". In: *LANGETI 2020 - Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing, Co-located with ESEC/FSE 2020* (Nov. 2020), pp. 7–12. DOI: 10.1145/3416504.3424334.

[46] Mohammad Mehedi Hassan and Akond Rahman. "As Code Testing: Characterizing Test Quality in Open Source Ansible Development". In: *Proceedings - 2022 IEEE 15th International Conference on Software Testing, Verification and Validation, ICST 2022* (2022), pp. 208–219. DOI: 10.1109/ICST53961.2022.00031.

[47] Andrew F. Hayes and Klaus Krippendorff. "Answering the Call for a Standard Reliability Measure for Coding Data". In: *Communication Methods and Measures* 1.1 (Apr. 2007), pp. 77–89. ISSN: 1931-2458. DOI: 10.1080/19312450709336664.

[48] Tjasa Hericko and Bostjan Sumak. "Analyzing Linter Usage and Warnings Through Mining Software Repositories: A Longitudinal Case Study of JavaScript Packages". In: *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology, MIPRO 2022 - Proceedings* (2022), pp. 1375–1380. DOI: 10.23919/MIPRO55190.2022.9803554.

[49] Mário Hozano et al. "Are you smelling it? Investigating how similar developers detect code smells". In: *Information and Software Technology* 93 (Jan. 2018), pp. 130–146. ISSN: 0950-5849. DOI: 10.1016/J.INFSOF.2017.09.002.

[50] *Infrastructure as Code Security — IaC Security — Snyk*. URL: https://snyk.io/product/infrastructure-as-code-security/.

[51] Ciera Christopher Jaspan, I. Chin Chen, and Anoop Sharma. "Understanding the value of program analysis tools". In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA* (2007), pp. 963–970. DOI: 10.1145/1297846.1297964.

[52] Karthick Jayaraman et al. *Automated Analysis and Debugging of Network Connectivity Policies*. Tech. rep. MSR-TR-2014-102. Microsoft, July 2014. URL: https:

//www.microsoft.com/en-us/research/publication/automated-analysis-and-debugging-of-network-connectivity-policies/.

[53] Brittany Johnson et al. "Why don't software developers use static analysis tools to find bugs?" In: *Proceedings - International Conference on Software Engineering* (2013), pp. 672–681. ISSN: 02705257. DOI: 10.1109/ICSE.2013.6606613.

[54] Stephen C Johnson. *Lint, a C Program Checker*. Tech. rep. 78-1273. Bell Labs, Oct. 1978. URL: https://web.archive.org/web/20220123141016/https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.1841&rep=rep1&type=pdf.

[55] Jai Kannan et al. "MLSmellHound". In: (May 2022), pp. 66–70. DOI: 10.1145/3510455.3512773.

[56] *KICS - Keeping Infrastructure as Code Secure*. URL: https://www.kics.io/.

[57] Indika Kumara et al. "The do's and don'ts of infrastructure code: A systematic gray literature review". In: *Information and Software Technology* 137 (Sept. 2021), p. 106593. ISSN: 0950-5849. DOI: 10.1016/J.INFSOF.2021.106593.

[58] Indika Kumara et al. "Towards Semantic Detection of Smells in Cloud Infrastructure Code". In: *ACM International Conference Proceeding Series* Part F162565 (2020), pp. 63–67. DOI: 10.1145/3405962.3405979.

[59] Julien Lepiller et al. "Analyzing Infrastructure as Code to Prevent Intra-update Sniping Vulnerabilities". In: (2021), pp. 105–123. ISSN: 1611-3349. DOI: 10.1007/978-3-030-72013-1{_}6.

[60] Hui Liu et al. "Deep learning based code smell detection". In: *IEEE Transactions on Software Engineering* 47.9 (Sept. 2021), pp. 1811–1837. ISSN: 19393520. DOI: 10.1109/TSE.2019.2936376.

[61] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. 2011. DOI: 10.6028/NIST.SP.800-145.

[62] Naouel Moha et al. "DECOR: A method for the specification and detection of code and design smells". In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 20–36. ISSN: 00985589. DOI: 10.1109/TSE.2009.50.

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

[64] Irineu Moura et al. "Mining energy-aware commits". In: *IEEE International Working Conference on Mining Software Repositories* 2015-August (Aug. 2015), pp. 56–67. ISSN: 21601860. DOI: 10.1109/MSR.2015.13.

[65] Allia Neamt. "From Terraform to AWS CloudFormation: A Study of Cost Patterns and Antipatterns". Bachelor's Thesis. University of Groningen, 2024.

[66] Evangelos Ntentos et al. "Detecting and Resolving Coupling-Related Infrastructure as Code Based Architecture Smells in Microservice Deployments". In: *IEEE International Conference on Cloud Computing, CLOUD* 2023-July (2023), pp. 201–211. ISSN: 21596190. DOI: 10.1109/CLOUD60044.2023.00031.

[67] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?" In: *Proceedings - 2023 IEEE/ACM 20th International Conference on Mining Software Repositories, MSR 2023* (2023), pp. 534–545. DOI: 10.1109/MSR59073.2023.00079.

[68] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime". In: *Proceed-*

ings - 2022 Mining Software Repositories Conference, MSR 2022 12.22 (2022), pp. 61–72. DOI: 10.1145/3524842.3527964.

[69] Eneko Osaba et al. "An Evolutionary Computation-Based Platform for Optimizing Infrastructure-as-Code Deployment Configurations". In: *Lecture Notes in Networks and Systems* 695 LNNS (2024), pp. 321–330. ISSN: 23673389. DOI: 10.1007/978-981-99-3043-2_25.

[70] *Overview — terraform-compliance*. URL: https://terraform-compliance.com/.

[71] Stefano Dalla Palma et al. "Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics". In: *IEEE Transactions on Software Engineering* 48.6 (June 2022), pp. 2086–2104. ISSN: 19393520. DOI: 10.1109/TSE.2021.3051492.

[72] Fabio Palomba et al. "Lightweight detection of Android-specific code smells: The aDoctor project". In: *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering* (Mar. 2017), pp. 487–491. DOI: 10.1109/SANER.2017.7884659.

[73] Fabiano Pecorelli et al. "Comparing heuristic and machine learning approaches for metric-based code smell detection". In: *IEEE International Conference on Program Comprehension* 2019-May (May 2019), pp. 93–104. DOI: 10.1109/ICPC.2019.00023.

[74] Willard Rafnsson et al. "Fixing Vulnerabilities Automatically with Linters". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12570 LNCS (2020), pp. 224–244. ISSN: 16113349. DOI: 10.1007/978-3-030-65745-1_13.

[75] Akond Rahman, Effat Farhana, and Laurie Williams. "The 'as code' activities: development anti-patterns for infrastructure as code". In: *Empirical Software Engineering* 25.5 (Sept. 2020), pp. 3430–3467. ISSN: 15737616. DOI: 10.1007/S10664-020-09841-8/TABLES/31.

[76] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. "A systematic mapping study of infrastructure as code research". In: *Information and Software Technology* 108 (Apr. 2019), pp. 65–77. ISSN: 0950-5849. DOI: 10.1016/J.INFSOF.2018.12.004.

[77] Akond Rahman, Chris Parnin, and Laurie Williams. "The Seven Sins: Security Smells in Infrastructure as Code Scripts". In: *Proceedings - International Conference on Software Engineering* 2019-May (May 2019), pp. 164–175. ISSN: 02705257. DOI: 10.1109/ICSE.2019.00033.

[78] Akond Rahman and Laurie Williams. "Source code properties of defective infrastructure as code scripts". In: *Information and Software Technology* 112 (Aug. 2019), pp. 148–163. ISSN: 0950-5849. DOI: 10.1016/J.INFSOF.2019.04.013.

[79] Akond Rahman et al. "Bugs in Infrastructure as Code". In: (Sept. 2018). URL: https://arxiv.org/abs/1809.07937v2.

[80] Akond Rahman et al. "Gang of eight: A defect taxonomy for infrastructure as code scripts". In: *Proceedings - International Conference on Software Engineering* 13.20 (June 2020), pp. 752–764. ISSN: 02705257. DOI: 10.1145/3377811.3380409.

[81] Akond Rahman et al. "Security Smells in Ansible and Chef Scripts". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30.1 (Mar. 2021), p. 3. ISSN: 15577392. DOI: 10.1145/3408897.

[82] Ghulam Rasool and Azhar Ali. “Recovering Android Bad Smells from Android Applications”. In: *Arabian Journal for Science and Engineering* 45.4 (Apr. 2020), pp. 3289–3315. ISSN: 21914281. DOI: 10.1007/S13369-020-04365-1.

[83] Pandu Ranga Reddy Konala, Vimal Kumar, and David Bainbridge. “SoK: Static Configuration Analysis in Infrastructure as Code Scripts”. In: *Proceedings of the 2023 IEEE International Conference on Cyber Security and Resilience, CSR 2023* (2023), pp. 281–288. DOI: 10.1109/CSR57506.2023.10224925.

[84] *Regula*. URL: https://regula.dev/.

[85] Yeonhee Ryou et al. “Code Understanding Linter to Detect Variable Misuse”. In: *ACM International Conference Proceeding Series* (Sept. 2022). DOI: 10.1145/3551349.3559497.

[86] Nuno Saavedra and João F. Ferreira. “GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code”. In: *ACM International Conference Proceeding Series* (Sept. 2022). DOI: 10.1145/3551349.3556945.

[87] José Amancio M. Santos et al. “A systematic review on the code smell effect”. In: *Journal of Systems and Software* 144 (Oct. 2018), pp. 450–477. ISSN: 0164-1212. DOI: 10.1016/J.JSS.2018.07.035.

[88] Jan Schumacher et al. “Building empirical support for automated code smell detection”. In: *ESEM 2010 - Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (2010). DOI: 10.1145/1852786.1852797.

[89] Julian Schwarz, Andreas Steffens, and Horst Lichter. “Code smells in infrastructure as code”. In: *Proceedings - 2018 International Conference on the Quality of Information and Communications Technology, QUATIC 2018* (Dec. 2018), pp. 220–228. DOI: 10.1109/QUATIC.2018.00040.

[90] *semgrep/semgrep: Lightweight static analysis for many languages. Find bug variants with patterns that look like source code.* URL: https://github.com/semgrep/semgrep.

[91] Rian Shambaugh, Aaron Weiss, and Arjun Guha. “Rehearsal: A configuration verification tool for puppet”. In: *ACM SIGPLAN Notices* 51.6 (June 2016), pp. 416–430. ISSN: 15232867. DOI: 10.1145/2908080.2908083.

[92] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. “Does your configuration code smell?” In: *Proceedings - 13th Working Conference on Mining Software Repositories, MSR 2016* (May 2016), pp. 189–200. DOI: 10.1145/2901739.2901761.

[93] *SonarLint for IntelliJ*. URL: https://docs.sonarsource.com/sonarlint/intellij/.

[94] *tenable/terrascan: Detect compliance and security violations across Infrastructure as Code to mitigate risk before provisioning cloud native infrastructure.* URL: https://github.com/tenable/terrascan.

[95] *terraform-linters/tflint: A Pluggable Terraform Linter*. URL: https://github.com/terraform-linters/tflint.

[96] Kristin Fjola Tomasdottir, Mauricio Aniche, and Arie Van Deursen. “The Adoption of JavaScript Linters in Practice: A Case Study on ESLint”. In: *IEEE Transactions on Software Engineering* 46.8 (Aug. 2020), pp. 863–891. ISSN: 19393520. DOI: 10.1109/TSE.2018.2871058.

[97] Kristin Fjola Tomasdottir, Mauricio Aniche, and Arie Van Deursen. "Why and how JavaScript developers use linters". In: *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Nov. 2017), pp. 578–589. DOI: 10.1109/ASE.2017.8115668.

[98] *TOP IDE index*. URL: https://pypl.github.io/IDE.html.

[99] Yuki Ueda, Takashi Ishio, and Kenichi Matsumoto. "DevReplay: Linter that generates regular expressions for repeating code changes". In: *Science of Computer Programming* 223 (Nov. 2022), p. 102857. ISSN: 0167-6423. DOI: 10.1016/J.SCICO.2022.102857.

[100] *Use Bicep linter - Azure Resource Manager — Microsoft Learn*. URL: https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/linter.

[101] Carmine Vassallo et al. "Configuration smells in continuous delivery pipelines: A linter and a six-month study on GitLab". In: *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Nov. 2020), pp. 327–337. DOI: 10.1145/3368089.3409709.

[102] Sevilay Velioglu and Yunus Emre Selcuk. "An automated code smell and anti-pattern detection approach". In: *Proceedings - 2017 15th IEEE/ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2017* (June 2017), pp. 271–275. DOI: 10.1109/SERA.2017.7965737.

[103] Andrew Walker, Dipta Das, and Tomas Cerny. "Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study". In: *Applied Sciences 2020, Vol. 10, Page 7800* 10.21 (Nov. 2020), p. 7800. ISSN: 2076-3417. DOI: 10.3390/APP10217800.

[104] *wayfair-archive/terrafirma: A static analysis tool for Terraform plans*. URL: https://github.com/wayfair-archive/terrafirma.

[105] Fadi Wedyan, Dalal Alrmuny, and James M. Bieman. "The effectiveness of automated static analysis tools for fault detection and refactoring prediction". In: *Proceedings - 2nd International Conference on Software Testing, Verification, and Validation, ICST 2009* (2009), pp. 141–150. DOI: 10.1109/ICST.2009.21.

[106] Claes Wohlin et al. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012. ISBN: 9783642290442. DOI: 10.1007/978-3-642-29044-2.

A List of (Anti)pattern Occurrences

Budget

- AJarombek/global-aws-infrastructure (4a89f4b)
- MartinFeineis/terraform (359ba42)
- stuartellis/stuartellis-org-tf-modules (39a9cab)
- forgotpw/forgotpw-infrastructure (f4363ad)
- Katesagay/terraform-repo (9aacfbe)
- darogina/terragrunt-aws-modules (9c84d03)
- chetanbothra/Terraform_AWS_Billing_Alert (43b0d3b)
- ntk1000/aws-terraform-template (d016b96)
- StratusGrid/terraform-aws-single-account-starter (c291c09)
- nsbno/terraform-aws-cost-alarm (7e13549)
- 64kramsystem/ultimate_aws_certified_cloud_practitioner_course_terraform_configuration (2f36b8a)
- kelledge/idkfa (25cda0b)
- oke-py/aws-tf (ec2982c)
- cob16/aws_static_website (0d4fbd0)
- mintak21/terraform-old (c10b476)
- alphagov/govwifi-terraform (348b52a)
- eladidan/speedyhead.xyz-terraform (71f034f)
- coremaker/terraform-google-nucleus (11234f6)
- robgmills/jumpbox (028bbe1)
- AdrianNeatu/blog-terraform (3ba302c)
- richardhughes/infra-modules (48015a8)
- patheard/terraform-cantrill-aws-associate (bc7484c)
- alghanmi/terraform-modules (f4e8069)
- cds-snc/cloud-based-sensor (10bb572)
- rmaheshvarma/terraform (c866a7d)
- akerl/aws-account (91967d4)
- singaporewaketools/iaac (197502b)

Spot instances

- openinfrastructure/terraform-google-gitlab-runner (8429375)
- kathputli/terraform-aws (321b1ae)
- naciriii/terraform-ec2-gitlab-runner (f8af6bc)

- Hapag-Lloyd/terraform-aws-bastion-host-ssm (516075e)
- ToruMakabe/aks-anti-dry-iac (4ba7a9d)
- paperphyte/terraform-drone (79f4b7c)
- filhodanuvem/from-dev-to-ops (998be81)
- stephaneclavel/terraform (74b4ba4)
- tale-toul/SingleNodeOpenshiftOnLibvirt (6384306)
- JaredStufftGD/grok-airflow (7ac9544)
- kaz/kiritan.com (1cd96c7)
- openinfrastructure/terraform-google-multinic (7a9c468)

Object storage lifecycle rules

- alphagov/govuk-aws (f844cd8)
- alphagov/govuk-terraform-provisioning (ac105ab)
- ExpediaGroup/apiary-data-lake (47e62f2)
- SamTowne/BasketballDrillBot (4ec6d54)
- utilitywarehouse/tf_telecom (1700745)
- trajano/terraform-s3-backend (cb9f00a)

Expensive instance

- beaulabs/terraform_aws_ec2_instance (d6df68d)
- gudlyf/TerraformOpenVPN (4bc861c)
- IncredibleHolg/infra-aws-code (7090470)
- ministryofjustice/cloud-platform-infrastructure (e5dd13d)
- cisagov/cyhy_amis (4e67a50)
- Kalmalyzer/UE-Jenkins-BuildSystem (6360975)
- dshmelev/aws_kube_tc (853298a)
- aaaaasam/azure (c7bc0ce)
- rbabyuk/terra (beae899)
- Leonard-Ta/Sample-Security-service-Terraform (c16481a)
- jjffggpp/jjffggpp (93ee12a)
- EngineerBetter/kf-infra (fa5f7fb)
- KoutaroNohira/hashicat (81dc1d3)
- wallnerryan/terraform-scaleio (605e74f)
- UrbanOS-Examples/common (206394b)
- cookpad/terraform-aws-eks (59c4028)
- ayltai/hknews-infrastructure (68171be)
- joelchrist/terraform (bbf18d6)
- jg210/aws-experiments (5ff37f1)
- scott45/vof-deployment-scripts (c6b2c1b)
- fdns/terraform-k8s (f106917)
- ministryofjustice/hmpps-env-configs (670c006)
- ministryofjustice/hmpps-env-configs (954dda6)
- Linaro/qa-reports.linaro.org (76c8d1e)
- KieniL/terraform_setups (37f66bc)

- digio/terraform-google-gitlab-runner (07f8279)
- jharley/azure-basic-demo (7cd3d20)
- pangeo-data/terraform-deploy (f8163bd)
- pangeo-data/terraform-deploy (7244eed)
- aeternity/terraform-aws-devnet (f4113a8)
- schubergphilis/terraform-aws-mcaf-matillion (3b0e2fe)
- binbashar/le-tf-infra-aws (0208ae3)
- fpco/terraform-aws-foundation (cfe9203)
- Civil-Service-Human-Resources/lpg-terraform-paas (59477d3)
- ibm-cloud-architecture/iks_vpc_lab (629819c)
- goodpen/gke-v.1.0 (45053a0)
- rshurts/gke-cd-with-spinnaker (3bc712a)
- kaz/kiritan.com (1cd96c7)
- midl-dev/tezos-auxiliary-cluster (9cbfeba)
- NLnetLabs/rpki-deploy (8bd6e74)
- TimonB/tf-azure-example (ce89df3)
- 00inboxtest/terraform-google-vault (1d0b5db)
- Amberoat/didactic-octo-eureka (494706f)
- robertdebock/terraform-aws-vault (757edca)
- covid-videoplattform/covid-videoplattform (83d8b92)
- alphagov/govuk-infrastructure (a51a3bf)
- pelias/terraform-elasticsearch (8454c8e)
- ironpeakservices/infrastructure (2ca24fa)
- robertdebock/git-terraform-demo (5638b1a)
- robertdebock/git-terraform-demo (6863740)
- jenkins-infra/aws (586fde0)
- poseidon/terraform-azure-kubernetes (4989bf2)
- poseidon/typhoon (8d2c8b8)
- poseidon/typhoon (b68f8bb)
- tlc-pack/ci-terraform (af285dd)
- binbashar/le-tf-infra-aws (10cf135)

Old generation

- gudlyf/TerraformOpenVPN (be1245d)
- alphagov/govuk-aws (6cfda6a)
- alphagov/govuk-aws (aeb3bfb)
- alphagov/govuk-aws (5fa5da9)
- alphagov/govuk-aws (19d187e)
- alphagov/govuk-aws (806b1a2)
- alphagov/govuk-aws (8d7d2eb)
- greenbrian/musical-spork (24c07bf)
- dotancohen81/Rancher (9094427)
- cisagov/cyhy_amis (7b8d924)
- yardbirdsax/elasticsearch-the-hard-way (521bae5)
- GBergeret/tf-vpc-module (34d80ec)

- cisagov/vulnerable-instances (f704100)
- dwp/dataworks-aws-data-egress (14f065e)
- circleci/enterprise-setup (26cc529)
- bh1m2rn/gitlab-environment-toolkit (b9750f0)
- travis-ci/terraform-config (4f641b1)
- byu-oit/terraform-aws-rds (86a0795)
- poseidon/terraform-azure-kubernetes (633eb93)
- poseidon/terraform-aws-kubernetes (e09126b)
- deadlysyn/terraform-keycloak-aws (1c982ac)
- ONSdigital/eq-terraform (79845fe)
- kinvolk-archives/lokomotive-kubernetes (f2f4deb)
- smarman85/a_new_hope (de97a6b)
- kmishra9/PL2-AWS-Setup (0d7b5b0)
- cisagov/cool-sharedservices-nessus (5403a89)
- guillaumekh/wg-terraform-template (effee9c)
- ninthnails/terraform-aws-camellia (0019704)
- openaustralia/infrastructure (63ee190)
- robertdebock/terraform-aws-vault (e3b6520)
- cisagov/cool-assessment-terraform (3138943)
- pelias/terraform-elasticsearch (21c1827)
- lowflying/OVPN—TF (be1245d)
- figurate/bedrock (bffc023)
- alphagov/govuk-aws (ffa7525)
- jg210/aws-experiments (b09b668)
- ministryofjustice/cloud-platform-environments (ce50204)
- ministryofjustice/cloud-platform-environments (b6cea25)
- ministryofjustice/cloud-platform-environments (aa07f2d)

Expensive storage type

- thomastodon/jabujabu (02210a3)
- giantswarm/giantnetes-terraform (53ed24b)
- Kalmalyzer/UE-Jenkins-BuildSystem (ee8942b)
- Leonard-Ta/Sample-Security-service-Terraform (c16481a)
- falldamagestudio/UE4-GHA-BuildSystem (e58083a)
- bculberson/btc2snowflake (9f8227b)
- ministryofjustice/hmpps-env-configs (0328838)
- travis-infrastructure/terraform-stuff (1e208af)
- sdcote/cloudsql (dfe44fc)
- wellcomecollection/archivematica-infrastructure (ce576be)
- jshcmpbll/Cloud-Mac-KVM (361885d)
- TimonB/tf-azure-example (b49579f)
- phillhocking/aws-ubuntu-irssi (1532e0c)
- bhfsystem/fogg (81e606a)
- bhfsystem/fogg (7cc487f)
- cisagov/freeipa-server-tf-module (99fd319)

- ministryofjustice/cloud-platform-terraform-monitoring (87401ba)
- ministryofjustice/hmpps-env-configs (7c1ba78)

Expensive network resource

- stealthHat/k8s-terraform (681a3f8)
- thomastodon/jabujabu (02210a3)
- structurefall/jamulus-builder (7190744)
- joshuaspence/infrastructure (d8e1979)
- joshuaspence/infrastructure (b9b9465)
- dexterchan/Terraform_Webserver (af5af0b)
- austin1237/clip-stitcher (4eed76f)
- IoT-Data-Marketplace/mp-infrastructure (5afcf39)
- InvictrixRom/website-infrastructure (09e4004)
- InvictrixRom/website-infrastructure (44d6632)
- pvandervelde/infrastructure.azure.core.network.hub (0ecf0a1)
- Midas-Protocol/webtwo-infra (25ed031)
- GBergeret/tf-vpc-module (5e63c83)
- GBergeret/micro-service-as-code (46f76d5)
- ecsworkshop2018/expertalk-2018-ecs-workshop (034908d)
- kitchen/personal-terraform (fe1f266)
- robertlupinek/rh-ex407 (0c679d7)
- skehlet/aws-batch-processing (decdbce)
- poseidon/terraform-aws-kubernetes (ef0372d)
- paperphyte/terraform-drone (f62bfeb)
- nisunisu/AWS_Blue_Green_Deployment (d0741cd)
- ryanlg/ryhino-public (e51b958)
- masterpointio/terraform-aws-nuke-bomber (33fbb76)
- kinvolk-archives/lokomotive-kubernetes (0c4d59d)
- alhardy-net/terraform-core-aws-alhardynet-networking (30be6aa)
- alhardy-net/terraform-core-aws-alhardynet-networking (f7b96f0)
- alhardy-net/terraform-core-aws-alhardynet-networking (b26b9e5)
- imma/fogg-env (7de4530)
- schubergphilis/terraform-aws-mcaf-matillion (3b0e2fe)
- binbashar/le-tf-infra-aws (a873443)
- binbashar/le-tf-infra-aws (19c37f7)
- binbashar/le-tf-infra-aws (bbfbd24)
- mads-hartmann/cloud.mads-hartmann.com (667f571)
- opszero/terraform-aws-kubespot (decc970)
- stSoftwareAU/sts-network (bf59a4c)
- lean-delivery/terraform-module-aws-core (25bbff7)
- CheesecakeLabs/django-drf-boilerplate (e4003aa)
- covidapihub/terraform-covidapihub (3c5d381)
- simplygenius/atmos-recipes (d27b483)
- hellupline/terraform-eks-cluster (2bd0135)
- tsub/ecs-sandbox (8501faf)

- schubergphilis/terraform-aws-mcaf-vpc (6ca41e5)
- davidcallen/parkrunpointsleague (21627e4)
- HarsheshShah08/HS-Terraform (e0d0f04)
- nagpach/terraform-example-aws-vpc (35d26fd)
- appbricks/cloud-inceptor (782a0a3)
- firehawkvfx/firehawk-prototype-deprecated (894fb1d)
- Xin00163/terraform (f69ce38)
- jeffawang/infrastructure (9f61081)
- stephengrier/my-infra (e5742d6)
- naftulikay/titan (a0ea4fd)
- alphagov/govuk-aws (9b54cd6)
- binbashar/le-tf-infra-aws (8b1c39c)
- robertdebock/terraform-aws-vault (4ddd021)

Overprovisioned resources

- thomastodon/jabujabu (02210a3)
- guilhermerenew/infra-cost (ba858d9)
- chaspy/terraform-alibaba-isucon8 (53588da)
- dwp/dataworks-aws-data-egress (14f065e)
- akaron/kubeadm_aws (2e2092e)
- robertdebock/terraform-azurerm-container-group (c0d6578)
- fdns/terraform-k8s (f106917)
- jackofallops/terraform-aws-mysql-cluster (7b2a446)
- alphagov/govwifi-terraform (38d0a67)
- pangeo-data/terraform-deploy (f8163bd)
- eduardobaitello/terraform-eks (c11fca6)
- jshcmpbll/Cloud-Mac-KVM (361885d)
- kaz/kiritan.com (1cd96c7)
- dylanmtaylor/dylanmtaylor-terraform-aws (44016d6)
- roysjosh/terraform-unifi (da9e286)
- phillhocking/aws-ubuntu-irssi (1532e0c)
- ministryofjustice/cloud-platform-terraform-monitoring (87401ba)
- wellcomecollection/buildkite-infrastructure (50957e0)

AWS - Expensive DynamoDB

- deptno/terraform-aws-modules (49f447b)
- ONSdigital/eq-terraform-dynamodb (40eb651)
- olliefr/aws-terraform-cloud1 (bf75383)
- garylb2/terraform-example-patterns (6de6d83)
- Arkoprabho/TerraformTutorial (ba317d7)
- jkstenzel95/jks.gameservers (411ab99)
- techservicesillinois/aws-enterprise-vpc (0d21bea)
- austin1237/gifbot (c11dabf)
- servers-tf/infrastructure (cc9e50a)

- jsoconno/aws-terraform-remote-state-infrastructure (fed8be2)
- ONSdigital/eq-terraform (6eaf697)
- nikkiwritescode/flask-app-terraform-deployment (af47bb6)
- kperson/terraform-modules (53bd2d8)
- kody-abe/terraform (169c776)
- jenkins-x/terraform-aws-eks-jx (cce6b14)
- telia-oss/terraform-aws-terraform-init (e8c7b2e)
- trajano/terraform-s3-backend (f4b61c7)
- poldi2015/chat-app (cb45bf1)
- tesera/terraform-modules (3cd4d7b)
- MichaelDeCorte/TerraForm (3799ee8)
- TalkingFox/SignalWs (935d9d6)
- codequest-eu/terraform-modules (ffe23d4)
- dgorbov/terraform-s3-backend-setup (81f8274)
- sbogacz/terraform-aws-state-backend (1744863)
- giuseppeborgese/terraform-locking-s3-state (6b4e59e)
- Accurate0/infrastructure (eef88fd)
- ministryofjustice/cloud-platform-environments (0c1b402)

Expensive monitoring

- Eximchain/terraform-aws-quorum-cluster (6a56f40)
- Accurate0/infrastructure (06889e0)
- cloudspout/Gefjun (665692a)
- terraform-google-modules/terraform-example-foundation (8391f1b)
- tdooner/flynn (a9ea9d0)
- elliotpryde/personal-infrastructure (772c5ad)
- elliotpryde/personal-infrastructure (7c4205c)
- alghanmi/terraform-modules (570d3d8)
- thoughtbot/flightdeck (c784bc0)
- chapas/tf-az-kubernetes (bcc6e19)
- Accurate0/infrastructure (bd63efe)
- rust-lang/simpleinfra (05370cc)
- binbashar/le-tf-infra-aws (8b1c39c)
- dfds/infrastructure-modules (1c9c92d)
- matihost/monorepo (6995c99)
- alphagov/govuk-infrastructure (3386d76)
- alphagov/govuk-infrastructure (6017d0b)

B TFLint Inspection Process

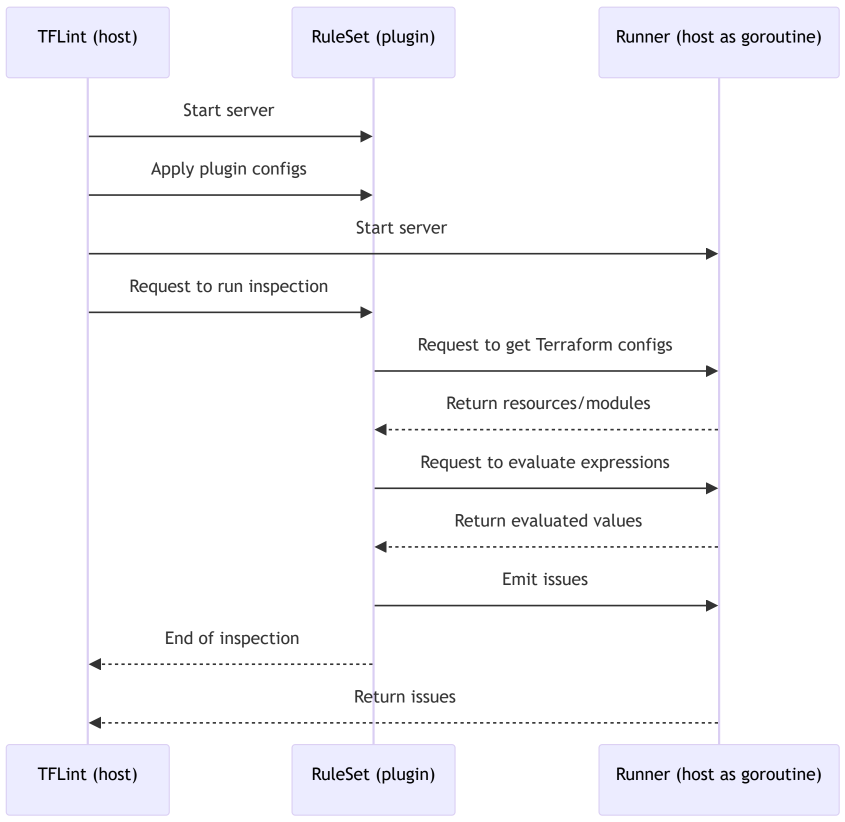


Figure B.1: Sequence diagram of TFLint's inspection process, from the TFLint documentation [1]

[1] https://github.com/terraform-linters/tflint/blob/master/docs/developer-guide/architecture.md

C Distribution of Cloud Providers

Figure C.1 lists the distribution of cloud provider codes assigned to commits. The total number of occurrences (568) does not match the number of commits (567); this difference is accounted for by the fact that 3 commits were assigned 2 cloud provider codes because of provider migrations and 3 commits were assigned 2 or more codes due to multi-cloud deployments. In addition, 6 commits were not assigned a cloud provider code because the commit did not apply to any specific cloud provider, instead using Terraform to e.g. apply Helm charts.

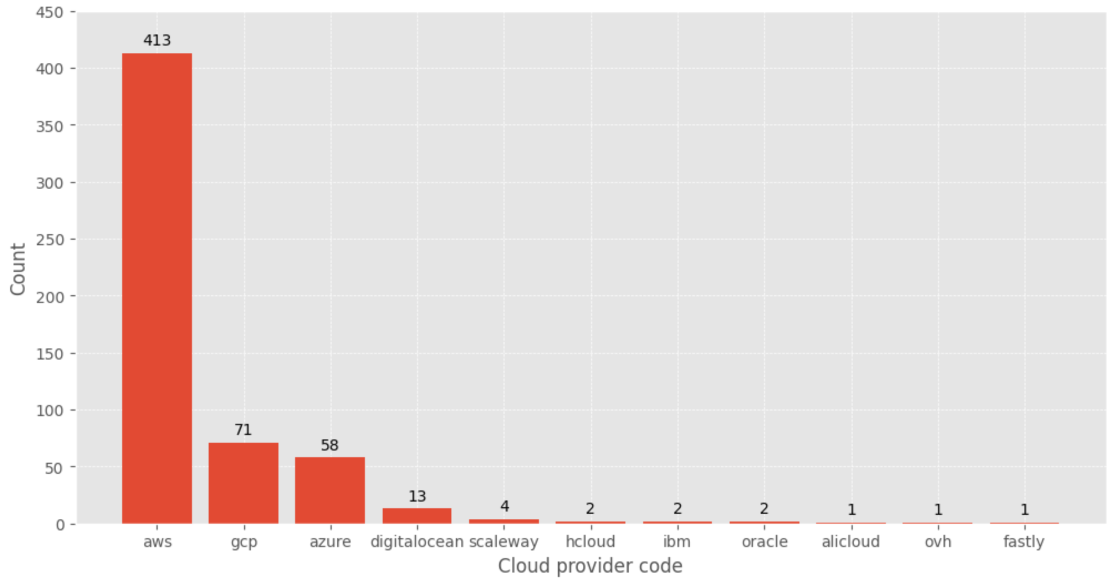


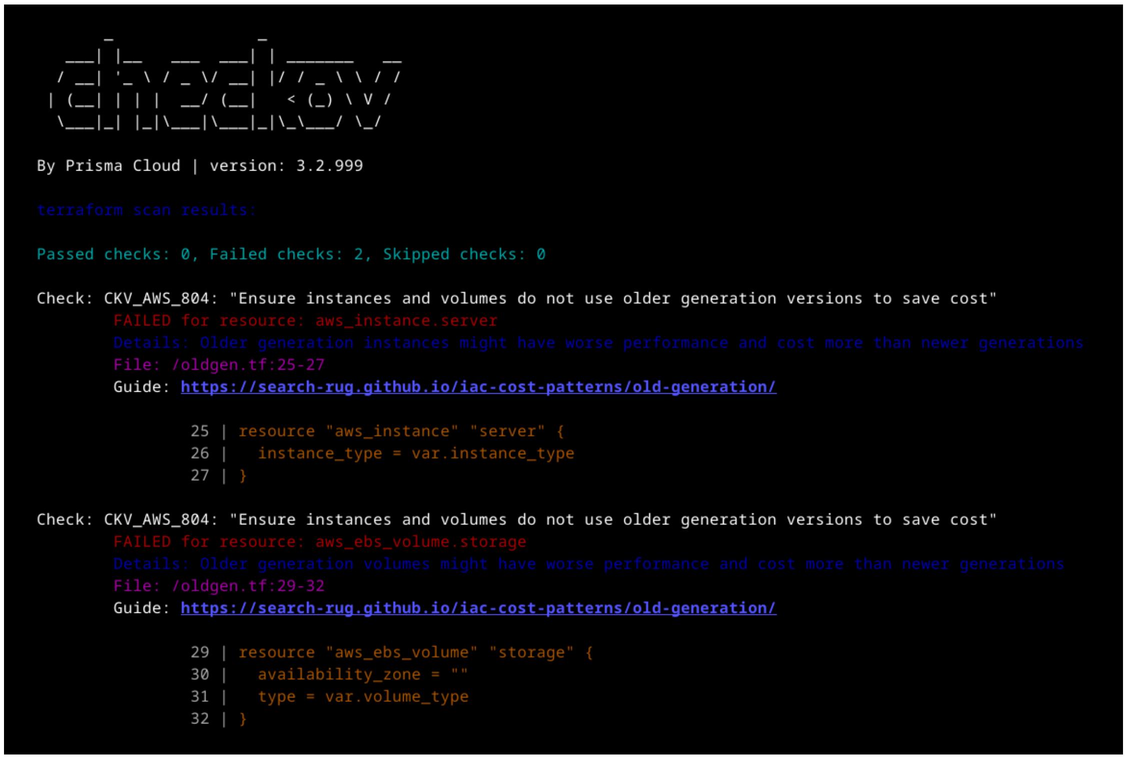
Figure C.1: Distribution of cloud provider codes

D Example Scans

```
resource "aws_instance" "server" {
  instance_type = "t2.micro"
}

resource "aws_ebs_volume" "storage" {
  availability_zone = ""
  type = "gp2"
}
```

Listing D.1: Example Terraform file for the *Old generation* antipattern



```
       _               _
   ___| |__   ___  ___| | _______   __
  / __| '_ \ / _ \/ __| |/ / _ \ \ / /
 | (__| | | |  __/ (__|   < (_) \ V /
  \___|_| |_|\___|\___|_|\_\___/ \_/

By Prisma Cloud | version: 3.2.999

terraform scan results:

Passed checks: 0, Failed checks: 2, Skipped checks: 0

Check: CKV_AWS_804: "Ensure instances and volumes do not use older generation versions to save cost"
        FAILED for resource: aws_instance.server
        Details: Older generation instances might have worse performance and cost more than newer generations
        File: /oldgen.tf:25-27
        Guide: https://search-rug.github.io/iac-cost-patterns/old-generation/

                25 | resource "aws_instance" "server" {
                26 |   instance_type = var.instance_type
                27 | }

Check: CKV_AWS_804: "Ensure instances and volumes do not use older generation versions to save cost"
        FAILED for resource: aws_ebs_volume.storage
        Details: Older generation volumes might have worse performance and cost more than newer generations
        File: /oldgen.tf:29-32
        Guide: https://search-rug.github.io/iac-cost-patterns/old-generation/

                29 | resource "aws_ebs_volume" "storage" {
                30 |   availability_zone = ""
                31 |   type = var.volume_type
                32 | }
```

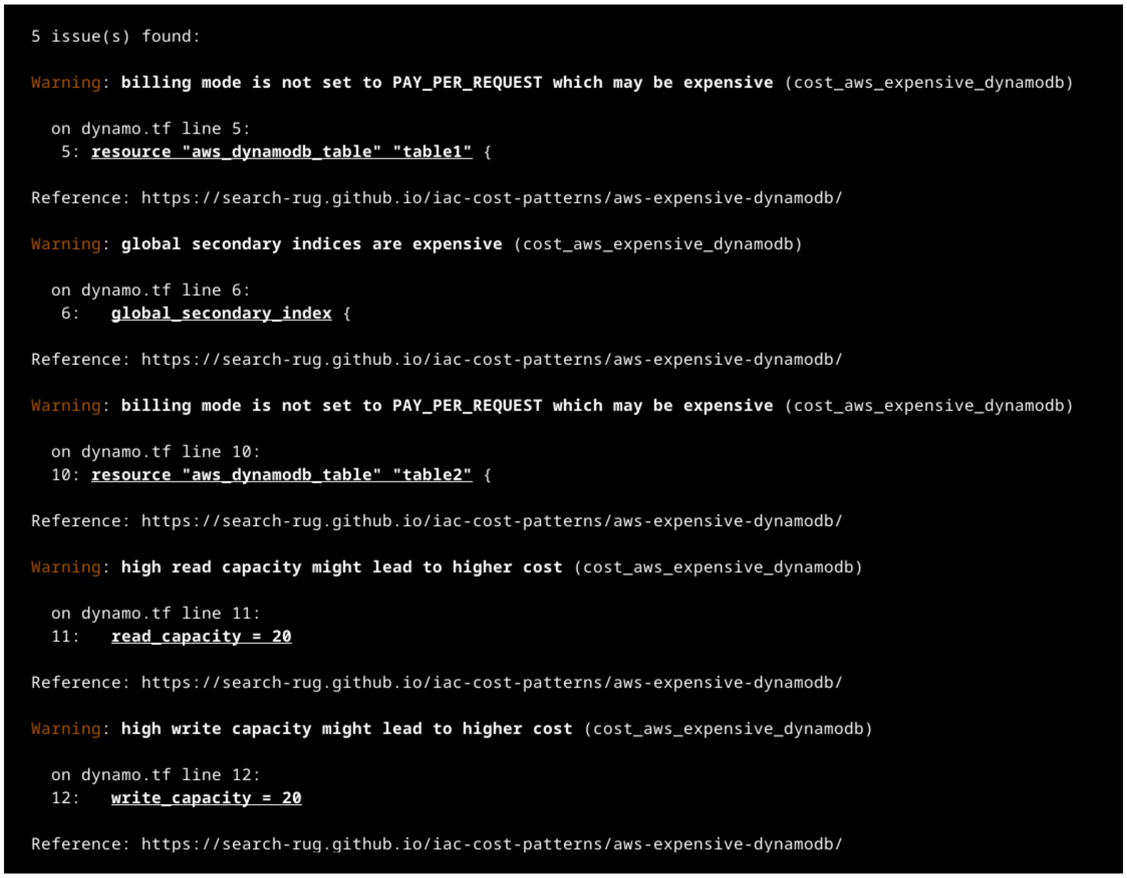
Figure D.1: Example result of a Checkov scan

```
resource "aws_dynamodb_table" "table0" {
  billing_mode = "PAY_PER_REQUEST"
}

resource "aws_dynamodb_table" "table1" {
  global_secondary_index {
  }
}

resource "aws_dynamodb_table" "table2" {
  read_capacity = 20
  write_capacity = 20
}
```

Listing D.2: Example Terraform file for the *AWS* - *Expensive DynamoDB* antipattern



```
5 issue(s) found:

Warning: billing mode is not set to PAY_PER_REQUEST which may be expensive (cost_aws_expensive_dynamodb)

  on dynamo.tf line 5:
   5: resource "aws_dynamodb_table" "table1" {

Reference: https://search-rug.github.io/iac-cost-patterns/aws-expensive-dynamodb/

Warning: global secondary indices are expensive (cost_aws_expensive_dynamodb)

  on dynamo.tf line 6:
   6:   global_secondary_index {

Reference: https://search-rug.github.io/iac-cost-patterns/aws-expensive-dynamodb/

Warning: billing mode is not set to PAY_PER_REQUEST which may be expensive (cost_aws_expensive_dynamodb)

  on dynamo.tf line 10:
  10: resource "aws_dynamodb_table" "table2" {

Reference: https://search-rug.github.io/iac-cost-patterns/aws-expensive-dynamodb/

Warning: high read capacity might lead to higher cost (cost_aws_expensive_dynamodb)

  on dynamo.tf line 11:
  11:   read_capacity = 20

Reference: https://search-rug.github.io/iac-cost-patterns/aws-expensive-dynamodb/

Warning: high write capacity might lead to higher cost (cost_aws_expensive_dynamodb)

  on dynamo.tf line 12:
  12:   write_capacity = 20

Reference: https://search-rug.github.io/iac-cost-patterns/aws-expensive-dynamodb/
```

Figure D.2: Example result of a TFLint scan