Master Thesis

IU University of Applied Sciences

Study program: M.Sc. Computer Science

Static IaC Analysis – Bridging the Gap between Research and Practice

Nils Leger

Enrolment number: 92004204

Im Mühlengrund 3

76456 Kuppenheim

Supervisor: Prof. André Köhler

Date of submission: 2nd of October 2022

Abstract

Context: Infrastructure as code is one of the main pillars in DevOps adopted by many companies. Since each infrastructure as code (IaC) tool has its own domain-specific language (DSL), practitioners must learn the IaC tool-specific DSL. This poses the threat of misconfiguration and security flaws. Unit, integration, and end-to-end testing for infrastructure code are more challenging than for application code. Thus, static code analysis plays an essential role in IaC quality assurance.
Objective: Researchers investigated defects in IaC scripts in various research studies. The findings of these studies, however, only benefit practitioners if they are incorporated into static infrastructure code analyzers (SICAs). No prior work has studied the state-of-the-art static infrastructure code analyzers from both a practical and academic perspective. This work bridges the gap between research and the various static code analyzers developed by practitioners. Furthermore, it provides decision support for practitioners and researchers.
Methodology: Existing static infrastructure code analyzers are identified using a multivocal literature review (MLR) because no prior work has been done in the field of static infrastructure code analysis in formal literature that also considers informal literature. The identified tools are assessed via qualitative analysis. The decision support is developed via design science research.
Results: Practitioners and researchers have developed various static infrastructure code analysis tools. Since each IaC tool has its own DSL, static analyzers must be adapted to each IaC tool. While many static analysis tools exist for popular IaC tools like *Ansible* and *Terraform*, development for other IaC tools and categories like resource visualization remains a gap.
Conclusion: The main contribution of this work is the application of the multivocal literature review methodology, which allows the inclusion of grey literature, thereby identifying a large number of static infrastructure code analyzers which have been ignored in formal literature so far. Researchers may use the result of this work to focus their research on yet understudied research areas. Furthermore, they may use existing static code analyzers to incorporate their findings into those tools instead of reinventing the wheel. If they decide to create a new SICA, researchers may refer to other SICAs to learn about implementation approaches. Practitioners can use the *IaC Analyzer Decision Guide* to support the selection of tools for quality assurance of their infrastructure code.

Keywords: Infrastructure as Code, IaC tools, static code analysis, static infrastructure code analyzers, multivocal literature review, design science research

Table of Contents

I. List of Figures ... IV

II. List of Tables ... V

III. List of Code Listings ... VI

IV. List of Abbreviations ... VII

1. Introduction ... 1

2. Literature Review ... 4

2.1. Cloud Deployment and Service Models ... 4

2.2. DevOps and DevSecOps ... 5

2.3. Infrastructure as Code ... 6

2.4. Static Code Analysis ... 10

2.5. Prior Research on Static Infrastructure Code Analysis ... 12

3. Research Methodology ... 23

3.1. Identification of Public Cloud Providers ... 26

3.2. Identification of IaC Tools ... 27

3.2.1. Planning the Review ... 27

3.2.2. Conducting the Review ... 27

3.2.3. Data Extraction and Synthesis ... 28

3.3. Identification of Static Infrastructure Code Analyzers ... 29

3.3.1. Planning the Review ... 30

3.3.2. Conducting the Review ... 30

3.3.3. Data Extraction and Synthesis ... 31

3.4. Identification of Open-Source and Proprietary Infrastructure Repositories ... 32

3.4.1. Open-Source Repositories ... 32

3.4.2. Proprietary Repositories ... 33

3.5. Assessment of Static Infrastructure Code Analyzers ... 34

3.5.1. RQ 3.5.1: What are the Characteristics of the Identified SICAs? ... 34

3.5.2. RQ 3.5.2: How do Practitioners Assess the Usefulness of the Findings of the Identified SICAs? ... 36

3.5.3. RQ 3.5.3: How are the Identified SICAs Implemented? ... 38

3.5.4. RQ 3.5.4: Which IaC Tools and Use-Cases Lack SICAs? ... 39

3.6. Design and Implementation of the Static IaC Analyzer Decision Guide ... 39

3.6.1. Analysis ... 40

3.6.2. Design ... 41

3.6.3. Evaluation ... 41

3.6.4. Diffusion ... 41

3.7. Summary of the Applied Methodology ... 42

4. Research Findings ... 43

4.1. Identification of Public Cloud Providers ... 43

4.2. Identification of IaC Tools 44

4.2.1. RQ 3.2.1: How do Practitioners and Researchers Categorize IaC Tools? 44

4.2.2. RQ 3.2.2: What IaC Tools Exist? 44

4.3. Identification of Static Infrastructure Code Analyzers 50

4.3.1. RQ 3.3.1: How do Practitioners and Researchers Categorize SICAs? 50

4.3.2. RQ 3.3.2: What SICAs Exist? 51

4.4. Identification of Open-Source and Proprietary Infrastructure Repositories 55

4.4.1. Open-Source Repositories 55

4.4.2. Proprietary Repositories 57

4.5. Assessment of Static Infrastructure Code Analyzers 57

4.5.1. RQ 3.5.1: What are the Characteristics of the Identified SICAs? 58

4.5.2. RQ 3.5.2: How do Practitioners Assess the Usefulness of the Findings of the Identified SICAs? 63

4.5.3. RQ 3.5.3: How are the Identified SICAs Implemented? 64

4.5.4. RQ 3.5.4: Which IaC Tools and Use-Cases Lack SICAs? 66

4.6. Design and Implementation of the Static IaC Analyzer Decision Guide 68

4.6.1. Analysis 69

4.6.2. Design 71

4.6.3. Evaluation 76

4.6.4. Diffusion 79

4.6.5. Result 80

4.6.6. Future Work on the IaC Analyzer Decision Guide 83

5. Conclusion 84

V. Bibliography VIII

VI. Appendices XVII

Appendix A Informed Consent Declaration XVIII

Appendix B Usability Test Plan XIX

Appendix C Usability Test Script XXI

Appendix D Usability Test Report XXV

Appendix E Tool Output XXXVI

Appendix F SICA Assessment XXXVII

Appendix G YAML Schema XLII

VII. Glossary XLIII

VIII. Declaration of Authenticity XLIV

I. List of Figures

Figure 1. The Testing Pyramid becomes a Diamond for IaC Testing 10
Figure 2. Checkov Output For a Publicly Accessible S3 Bucket Defined in Terraform........ 16
Figure 3. Checkov Output For a Publicly Accessible S3 Bucket Defined in Terraform with Rule Suppression........ 17
Figure 4. This Work Follows Six Consecutive Phases........ 23
Figure 5. An Overview of the Typical MLR Steps 24
Figure 6. The Four Phases of Design Science Research........ 42
Figure 7. IaC Tools: Number of Sources per Year........ 46
Figure 8. IaC Tools: Number of Sources by Source Type 46
Figure 9. IaC Tools: Number of Sources by Search String........ 47
Figure 10. IaC Tools: Votes per Category........ 48
Figure 11. SICAs: Number of Sources per Year........ 51
Figure 12. SICAs: Number of Sources by Source Type 52
Figure 13. SICAs: Number of Sources by Search String 52
Figure 14. SICA Characteristics: IaC Tools Included in the Analyzed Meta-Tools........ 58
Figure 15. SICA Characteristics: IaC Tool Support 59
Figure 16. SICA Characteristics: File Support........ 60
Figure 17. SICA Characteristics: Categories........ 61
Figure 18. SICA Characteristics: Licenses 62
Figure 19. SICA Characteristics: Programming Languages for the Tool Implementation........ 64
Figure 20. SICA Characteristics: Languages Used for the Rule Implementation 65
Figure 21. tfsec Output when the Custom Check Fails........ 74
Figure 22. IaC Analyzer Decision Guide: CI/CD Pipeline 81
Figure 23. IaC Analyzer Decision Guide 81
Figure 24. IaC Analyzer Decision Guide: Details Section of cloudformation-guard........ 82

Figure D-1. Usability Test Session Finding: Sorting XXVI
Figure D-2. Usability Test Session Finding: Info Text........ XXVII
Figure D-3. Usability Test Session Finding: Info Texts for Unknown Terms XXVIII
Figure D-4. Usability Test Session Finding: Number of Identified SICAs........ XXX
Figure D-5. Usability Test Session Finding: Export Option XXXI
Figure D-6. Usability Test Session Finding: Feedback Button........ XXXII
Figure D-7. Usability Test Session Finding: Save Search Results........ XXXII
Figure D-8. Usability Test Session Finding: How-To Use the IaC Analyzer Decision Guide XXXIII
Figure E-1. SICA Characteristics: Tool Output XXXVI

II. List of Tables

Table 1. Infrastructure Testing Approaches 13
Table 2. Terraform Testing Approaches 14
Table 3. Ansible Testing Approaches 15
Table 4. Inclusion and Exclusion Criteria for the Formal Literature Review on IaC Tools 28
Table 5. Inclusion and Exclusion Criteria for the Grey Literature Review on IaC Tools 28
Table 6. Inclusion and Exclusion Criteria for the Identified IaC Tools 29
Table 7. Inclusion and Exclusion Criteria for the Formal Literature Review on SICAs 30
Table 8. Inclusion and Exclusion Criteria for the Grey Literature Review on SICAs 31
Table 9. Inclusion and Exclusion Criteria for the Identified SICAs 31
Table 10. Inclusion and Exclusion Criteria for the Selection of Open-Source Repositories 33
Table 11. Inclusion and Exclusion Criteria for the Selection of Proprietary Repositories From Capgemini 33
Table 12. Criteria for the Assessment of the Identified SICAs 36
Table 13. Criteria for the In-Depth Assessment of the Selected SICAs 38
Table 14. Criteria for the Analysis of the Implementation Details of the Identified SICAs 38
Table 15. The Five Most Popular Public Cloud Providers 43
Table 16. IaC Tools: Search Strings and Stopping Criteria 48
Table 17. The Selected IaC Tools 49
Table 18. SICAs: Search Strings and Stopping Criteria 53
Table 19. The Selected SICAs for an In-Depth Assessment 55
Table 20. Open-Source Repositories for the Selected IaC Tools 56
Table 21. Proprietary Repositories for the Selected IaC Tools 57
Table 22. SICA Characteristics: SICAs with Web Applications 62
Table 23. SICA Characteristics: Cohen’s Kappa Score, False Positive Rate, Fix Rate, and Execution Time of the Selected SICAs 63
Table 24. How Existing SICAs Cover the Use Cases Visualization, Cost, Misconfiguration, and Security for the Selected IaC Tools 68

Table D-1. Usability Test Session: Legend XXVI
Table D-2. Usability Test Session: Findings XXVI
Table D-3. Findings for Each Participant: Legend XXXV
Table D-4. Findings for Each Participant: Summary XXXV
Table F-1. SICA Assessment XXXVIII

III. List of Code Listings

Listing 1. S3 Bucket Terraform Configuration with Public Access 16
Listing 2. S3 Bucket Terraform Configuration with Public Access and a Comment to Suppress a Warning 17
Listing 3. Example of a Rule written in Rego 65
Listing 4. Example of a Behavior-Driven Development Rule for terraform-compliance 66
Listing 5. S3 Bucket Defined with Pulumi in Typescript 67
Listing 6. Custom tfsec Check to Enforce the Configuration of a Docker Image Tag in the Terraform Manifest 75

Listing G-1. YAML Schema XLII

IV. List of Abbreviations

ACL	Access-Control List
API	Application Programming Interface
BDD	Behavior-Driven Development
CDK	Cloud Development Kit
CG	Capgemini
CI/CD	Continuous Integration / Continuous Delivery
CLI	Command-Line Interface
CM	Configuration Management
DSL	Domain-Specific Language
DSR	Design Science Research
GL	Grey Literature
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
MLR	Multivocal Literature Review
NLP	Natural Language Processing
OPA	Open Policy Agent
PaaS	Platform as a Service
RQ	Research Question
SaaS	Software as a Service
SAST	Static Application Security Testing
SICA	Static Infrastructure Code Analyzer
SLAC	Security Linter for Ansible and Chef Scripts
SLIC	Security Linter for Infrastructure as Code
SLR	Systematic Literature Review
VC	Version Control
VS Code	Visual Studio Code

1. Introduction

"*A long time ago, in a datacenter far, far away, an ancient group of powerful beings known as 'sysadmins' used to deploy infrastructure manually. Every server, every database, every load balancer, and every bit of network configuration was created and managed by hand. It was a dark and fearful age: fear of downtime, fear of accidental misconfiguration, fear of slow and fragile deployments, and fear of what would happen if the sysadmins fell to the dark side (i.e., took a vacation).*"

~ Yevgeniy Brikman (Brikman, 2022, Preface)

In the **iron age** of IT operation (Morris, 2020), the infrastructure was configured manually, as Brikman (2022) described above. Over the years, this approach exposed significant disadvantages. The list of service interruptions, outages, and security breaches caused by manual misconfiguration is long. For instance, Wang (2022) once manually copied and pasted the command *shutdown* instead of *no shutdown* between two text documents, causing a critical service outage. Due to the fear of such outages, releases became slower and more painful. Ultimately, the release frequency decreased, possibly reducing a company's performance (Forsgren et al., 2018).

The industry is shifting from the **iron age** to the **cloud age**. This shift manifests in companies renting their infrastructure resources in the cloud instead of operating data centers (Brikman, 2022). This transition influences the work of operation teams. Instead of maintaining hardware in data centers, they configure resources in the cloud via software tools. The work of operation teams and development teams converges. DevOps unites the traditional silos of operation and development teams via changes to organizations' cultures. A technological approach to support this transition is infrastructure as code (IaC). IaC aims to overcome the **fear** of errors due to **manual changes** by creating **confidence via automation**. It is the practice of writing code to define, create, update, and destroy infrastructure automatically and is inspired by software development techniques (Brikman, 2022; Fowler, 2012, July 10; Morris, 2013, June 13, 2020; Wang, 2022). Thereby IaC drives *automation*, one of the four main pillars of DevOps (Wang, 2022).

IaC aims to automate infrastructure configuration, decreasing the risk of manual misconfiguration during deployments (Chiari et al., 2022). Nevertheless, the infrastructure code must still be manually created. Hence, infrastructure code is vulnerable to defects, code smells, and other quality issues (Chiari et al., 2022; A. Rahman, Elder, et al., 2018). Such misconfigurations must be identified and resolved to take advantage of the countless benefits of IaC. Since the infrastructure manifests are defined as code, they can be tested as is known from application testing. Several testing approaches for infrastructure code exist (Brikman, 2022, chapter 9; Morris, 2020, chapter 9). The most basic approach is static analysis, which analyzes the syntax and semantics of code without executing it (Chiari et al., 2022).

Guerriero et al. (2019) found that developers generally perceive infrastructure code testing as challenging. Their observation suggests analyzing the entire infrastructure code ecosystem. Still, the present work is limited to static analysis because it is easy to use and an appropriate entry point to infrastructure code testing (Brikman, 2022, chapter 9). The present work prefers to analyze static infrastructure code analysis in-depth instead of all testing approaches more broadly.

Practitioners (Brikman, 2022; Morris, 2020) and researchers (A. Rahman, 2018a; A. Rahman, Elder, et al., 2018) explained that the static infrastructure code analysis ecosystem is immature. Although **formal research** studies analyzed existing static infrastructure code analyzers (SICAs), they focused on tools developed in formal literature. In general, formal literature only occasionally mentions SICAs from informal literature. **Practitioner-oriented reports** like Luzar et al. (2021) also incorporated SICAs from informal literature. Nevertheless, they focused only on four IaC tools. Moreover, i**nformal literature** only compared SICAs for specific IaC tools (e.g., Ibnelbachyr (2021)). In summary, neither formal nor informal literature analyzed the entire SICA ecosystem up to the present. Ultimately, formal literature requires such a study to bridge the gap between research and the large SICA ecosystem developed in practice and research. It will benefit researchers by revealing areas for future research, collecting characteristics of existing SICAs, and recording the state of practice. It will also benefit practitioners by helping them to find suitable SICAs for their projects.

This work analyzes existing SICAs for the most popular IaC tools from formal and informal literature by conducting a multivocal literature review. This methodology is commonly used to investigate practitioner-oriented fields in computer science research. This work is the first in the field to apply this methodology. It makes the collected data available to researchers and practitioners in an appropriate format: a web application. Based on the described lack of a research study analyzing the state of practice and research of SICAs, the following three main research questions (RQs) guide this work:

- **RQ 1**: *Which static infrastructure code analyzers exist, and what are their characteristics?*
- **RQ 2**: *Which gaps exist in static infrastructure code analysis?*
- **RQ 3**: *How can knowledge about static infrastructure code analyzers be made accessible to practitioners and researchers?*

Before investigating the state of research, this work's terminology must be clarified. As will be explained in chapter 2, a large IaC ecosystem of tools, frameworks, and languages exist. In this work, all of these are referred to as *IaC tools*. Each of these IaC tools has different descriptions for its source code formats. Since this work does not compare IaC tools, how the specific source code formats are called is irrelevant. Thus, all artifacts are referred to as *infrastructure code* or *infrastructure manifests* (Morris, 2020). The infrastructure code can be subject to syntax checking, linting, security scanning, and many other practices, as explained in chapter 2. All these practices have something in common: they are performed statically and analyze infrastructure code. Hence, tools performing these checks are called *static infrastructure code analyzers (SICAs)*. Furthermore, all IaC tools and SICAs are written in italics to improve the readability of this work.

The remainder of this work is structured as follows. Chapter 2 explains the state of research on cloud deployment models and types, DevOps, DevSecOps, IaC, static application code analysis, and static infrastructure code analysis. In chapter 3, the research methodology is outlined. Chapter 4 identifies existing cloud providers, IaC tools, and SICAs, assesses the SICAs, and finally designs the *IaC Analyzer Decision Guide* for practitioners and researchers. A summary of the most important findings of this work is provided in chapter 5, together with implications for researchers and practitioners, limitations of this work, and recommendations for future research.

2. Literature Review

This section introduces background information on cloud deployment and service models, DevOps, DevSecOps, infrastructure as code, and static code analysis. Finally, it outlines related work in the research area of static code analysis for infrastructure code. Initially, cloud deployment and service models are explained since infrastructure as code is closely related to cloud computing.

2.1. Cloud Deployment and Service Models

In cloud computing, several service offerings and cloud types must be distinguished. These are also referred to as deployment models and service models. This differentiation has implications for the different IaC tools and SICAs. For instance, the types of resources an IaC tool manages are significantly affected by the particular cloud service model. To take a concrete example, Software as a Service (SaaS) offerings do not allow the deployment of load balancers. There are four major cloud deployment models, which are now shortly explained.

Private Cloud

A private cloud belongs to one single organization (Jamsa, 2022, chapter 1), which offers cloud services on private infrastructure (Lisdorf, 2021, chapter 1). Hence, it provides more security but is more expensive than a public cloud (Jamsa, 2022, chapter 1).

Public Cloud

Several companies (e.g., Amazon Web Services) offer public cloud services that multiple organizations can use. Cloud providers have several approaches to offering highly available services, for instance, having multiple availability zones per geographic region (availability zone and region are AWS vocabulary) (Jamsa, 2022, chapter 1). Compared to the private cloud, it may be less secure "because of its openness" and less expensive (Jamsa, 2022, chapter 1). Nevertheless, the public cloud developed enhanced "capabilities for keeping the public cloud private" (Lisdorf, 2021, chapter 1).

Community Cloud

A community cloud is neither used by one organization nor shared across multiple independent organizations. Instead, the organizations are related and have "shared concerns" (Jamsa, 2022, chapter 1). For instance, the cloud is shared by "schools within a university" (Jamsa, 2022, chapter 1).

Hybrid Cloud

As the name implies, a hybrid cloud combines two or more of the previously discussed models (Jamsa, 2022, chapter 1). According to Lisdorf (2021, chapter 1), this deployment model is popular since many companies run their applications on the public cloud and their own data centers. Still, this implementation does not align with the original definition of a public cloud (Lisdorf, 2021, chapter 1).

In addition, cloud offerings are also distinguished into three service types. At the same time, the boundaries between service models continue to become more and more blurry (Lisdorf, 2021, chapter 1).

Software as a Service (SaaS)
With SaaS, an entire application is hosted in the cloud, which the users access via a web browser. An example SaaS application is Salesforce (Jamsa, 2022, chapter 1). Users cannot write code but can only configure the application (Lisdorf, 2021, chapter 1). In that way, a company can outsource maintenance and administration and pay on demand (Jamsa, 2022, chapter 2).

Platform as a Service (PaaS)
PaaS is a platform for developers where they can develop and deploy their applications (Lisdorf, 2021, chapter 1). The cloud platform provider manages the underlying hardware, like operating systems and databases, which allows developers to focus on application development (Jamsa, 2022, chapter 1). With PaaS, developers can outsource the maintenance and provisioning of server hardware and operating systems which also allows fast up- or down-scaling (Jamsa, 2022, chapter 3).

Infrastructure as a Service (IaaS)
IaaS offers low-level resources like servers, storage, and network for developers (Jamsa, 2022, chapter 1). In comparison to PaaS, companies have more responsibility. They may accept this trade-off because of security requirements (Jamsa, 2022, chapter 4). Still, IaaS eliminates the need for data centers and allows pay-as-you-go fees and fast scaling (Jamsa, 2022, chapter 4).

Companies use cloud computing for their technological revolution by outsourcing data center operations and utilizing cloud services. Likewise, companies also seek cultural revolutions. **DevOps**, in particular, is an approach to such a cultural revolution.

2.2. DevOps and DevSecOps

DevOps describes the close collaboration of **Dev**elopment and **Op**eration**s** achieved by a cultural change and the breakup of silos (Halstenberg et al., 2020, 1p). It is defined as "the integration of development and operations into lean and agile practices" (Wilson, 2020). Mixed teams of operations staff and developers are an essential approach in DevOps (Halstenberg et al., 2020, p. 5). This mixed team approach is vital because it unites two departments that naturally have opposite goals (Halstenberg et al., 2020, p. 22). Development teams want to deliver new software quickly, whereas operation teams focus on stability (Halstenberg et al., 2020, p. 22). DevOps is neither standardized nor pure automation (Halstenberg et al., 2020, p. 14). It is also not one tool or a toolchain. DevOps is more than tooling. It is a cultural change. For instance, DevOps teams aspire to a blame-free culture (Halstenberg et al., 2020, p. 30).

DevOps can be summarized on a high level with its four pillars. Those main pillars of DevOps are often abbreviated as CAMS (Halstenberg et al., 2020, 6p):

- **C**(ulture): The organizational culture plays the central role in adopting DevOps and must be applied to all levels (management and development teams).
- **A**(utomation): Automating the software delivery process from development to deployment in production.
- **M**(easurement): Key figures must be measured to quantify the status and progress of projects.
- **S**(haring): Sharing is closely related to the cultural aspect of DevOps. It describes sharing information across the entire organization, not only in a team.

Forsgren et al. (2018) identified 24 key capabilities that significantly influence the software delivery performance of a company. These capabilities include version control, deployment automation, test automation, and shift left on security (Forsgren et al., 2018, p. 19), all of which appear in DevOps. That said, not only security aspects are shifted left (i.e., executed sooner in the development process). Shifting left applies to all activities and decisions (Halstenberg et al., 2020, p. 15). Since its first mention in 2009 (Halstenberg et al., 2020, p. 3), DevOps has been extended, for instance, by incorporating security processes, resulting in the term DevSecOps (**Dev**elopment + **Sec**urity + **Op**eration**s**) (Wilson, 2020). In DevSecOps, security is thought of right from the start. Therefore, it is often referred to as a “shift left” approach for security, as mentioned earlier (Forsgren et al., 2018, p. 113). Even though Halstenberg et al. (2020, p. 15) explain that shifting left applies to all activities and decisions in DevOps, and this would also apply to security aspects, DevSecOps emphasizes the security aspect.

Several concepts and tools are regularly used in DevOps. For instance, version control plays a key role. Both application and infrastructure code are stored in version control. Moreover, continuous integration, delivery, and deployment ensure that the software is constantly tested, built, and delivered. Even though those paradigms are often mentioned with application code, they can also be applied to **infrastructure code**, as the following chapter shows.

2.3. Infrastructure as Code

Infrastructure as Code is closely related to DevOps (Chiari et al., 2022). Wilson (2020, p. 157) goes one step beyond a pure relationship between DevOps and IaC and claims that IaC is an outcome of DevOps because DevOps applies software development practices to infrastructure code. Referring back to DevOps, IaC is necessary for the automation parts of CAMS (Wang, 2022, p. 9). Infrastructure as Code describes the practice of defining the infrastructure as code instead of manually deploying infrastructure resources (Halstenberg et al., 2020, p. 22). The term infrastructure as code unites several categories of tools, and each category specializes in a particular infrastructure aspect.

The main characteristic of infrastructure as code is that the infrastructure is defined as code. Several advantages result from this attribute. First, IaC reduces deployment times compared to manual deployments since it allows automation (Chiari et al., 2022, p. 218). Furthermore, IaC modularizes infrastructure, thereby reducing deployment risk, and shares information about infrastructure configuration, which is stored in version control. IaC allows auditing and analyzing the infrastructure code and speeds up resolving issues. Last, it allows developers to be independent of operations and thereby start deployments on their own (Brikman, 2022, chapter 1; Morris, 2020, chapter 1; Wang, 2022, pp. 19–23). The downside of IaC is that it requires developers or operations teams to learn one or more new IaC tools compared to manual deployments. IaC allows organizations to consider the security and compliance of applications right from the start of the development process, for instance, by practicing policy as code, also known as "*shift-left security testing*, or *static analysis of IaC*" (Wang, 2022, p. 208). Policies are rules that encode an organization's security requirements (Wang, 2022, p. 200). Policy as code helps development teams to comply with these policies. To do so, policy as code tools evaluate the infrastructure code against the specified policies. This aligns with the DevSecOps practice of security as code, which integrates security policies into the development process (Myrbakken & Colomo-Palacios, 2017, p. 25).

A large ecosystem of open-source and commercial IaC tools emerged. Understanding the different categories and implementations of IaC tools provides the background for understanding how static code analyzers can work with those tools. Brikman (2022) distinguishes five categories of IaC tools.

- **Ad hoc scripts** are written in a programming language like Python to replace repetitive, manual effort. For instance, Siebra et al. (2018) introduced deployment automation via Powershell DSC for Windows in their organization.
- **Configuration management** (CM) tools such as *Ansible* (Red Hat, n.d.) expect already existing hardware or virtual servers. They configure the "packages and attributes hosted on them" (Wang, 2022, p. 26).
- **Server templating** tools like *Packer* (HashiCorp, n.d.–b) create "machine images for application runtimes." For example, they create containers or servers with all the software and configuration pre-installed and -configured (Wang, 2022, p. 27).
- **Orchestration** tools, such as *Kubernetes* (Cloud Native Computing Foundation, n.d.–b), manage several other infrastructure components. These tools roll out updates, monitor the resources' health, scale the number of resources, distribute traffic, and allow resources to communicate with each other (Brikman, 2022, chapter 1). In this context, it is essential to clarify the terms used. *Kubernetes* itself is only a platform. However, the *Kubernetes* manifests used to define the *Kubernetes* resources are infrastructure as code.
- **Provisioning** tools such as *Terraform* (HashiCorp, n.d.–c) are used to create and manage infrastructure resources like servers, databases, and load balancers or manage *Kubernetes* resources for a public cloud or data center (Wang, 2022, p. 25).

Even though categorizing IaC tools into five distinct categories appears plausible, practice is more complicated since IaC tools can belong to more than one of those categories (Brikman, 2022;

HashiCorp, n.d.–c). Beyond the classification of Brikman (2022), Wang (2022) uses a similar classification approach but does not explicitly mention ad-hoc scripts and orchestration tools as separate categories. Still, she uses ad-hoc scripts in Python and categorizes the orchestration tool *Kubernetes* as a provisioning tool (Wang, 2022, p. 105). Hence, she also considers orchestration tools as provisioning tools. This observation suggests that the outlined categories are popular in practice. In addition to the classes of IaC tools mentioned above, package managers such as *Helm* (Cloud Native Computing Foundation, n.d.–a) support reusable infrastructure code (Doolittle & Blumen, 2022). Even though package managers do not fit directly into one of the five categories, they are still considered IaC tools in this work because they are used to write infrastructure code. *Helm*, for instance, is an abstraction level above *Kubernetes* manifests and utilizes its own templating language. Since the IaC tools of the different categories support different use-cases, it is common and often required to combine several IaC tools to cover the entire application development lifecycle, including deployment to production, in practice (Brikman, 2022; HashiCorp, n.d.–c). The resulting complexity of choosing suitable IaC tools has been recognized by researchers and addressed, for instance, by Özel et al. (2020).

The IaC tools can be further sub-classified. Some classification approaches work across IaC tool categories (e.g., imperative and declarative), and some only within a category (e.g., agent and agentless). Most configuration management tools follow the **mutable** infrastructure concept. Mutable infrastructure means that the IaC tools update existing infrastructure. In contrast, provisioning and server templating tools follow the **immutable** infrastructure concept. They often create new infrastructure, replacing the old infrastructure. **Imperative** (sometimes also **procedural**) IaC tools describe the steps to achieve the desired target state. In contrast, **declarative** tools define the desired target state, and the CM tool plans the required steps. For instance, *Ansible* is an imperative IaC tool, whereas *Puppet* is declarative. Furthermore, some IaC tools, like *Pulumi,* use **general-purpose programming languages**. Most other IaC tools (e.g., *Ansible*) use **domain-specific languages**. Configuration management tools can require a **master**[1] **server** (e.g., *Chef*) or work **masterless** (e.g., *Ansible*). Such a master server stores the infrastructure state. Moreover, configuration management tools can be distinguished based on whether they need an **agent** installed on each server. If they do not require such an agent, they are said to be **agentless** (e.g., *Ansible*). Such an IaC tool logs into the server (e.g., via secure shell (SSH)). Other tools (e.g., *Chef*) do require an agent. (Brikman, 2022, chapter 1)

Server templating tools can be distinguished based on the type of templates they create. For instance, they create Virtual Machine (VM) images like *Packer* (HashiCorp, n.d.–b). Others create container images, like *Docker* (Docker, n.d.). Orchestration tools can be either self-managed (e.g., hosting a *Kubernetes* cluster (Cloud Native Computing Foundation, n.d.–b)) or obtained from a cloud

[1] The author decided to use the term master although its usage has been replaced in many areas. For instance, the master branch on GitHub has been renamed to main. Nevertheless, in this use case, the terms master and masterless are still used.

provider (e.g., *Amazon Kubernetes Service* (Amazon Web Services, n.d.–a)). Provisioning tools are either specific to a cloud provider like *AWS CloudFormation* (Amazon Web Services, n.d.–d) or support various cloud providers as *Terraform* does (HashiCorp, n.d.–c). The characteristics of an IaC tool affect how static code analysis can be performed. For instance, as explained later, effectively testing declarative code is challenging.

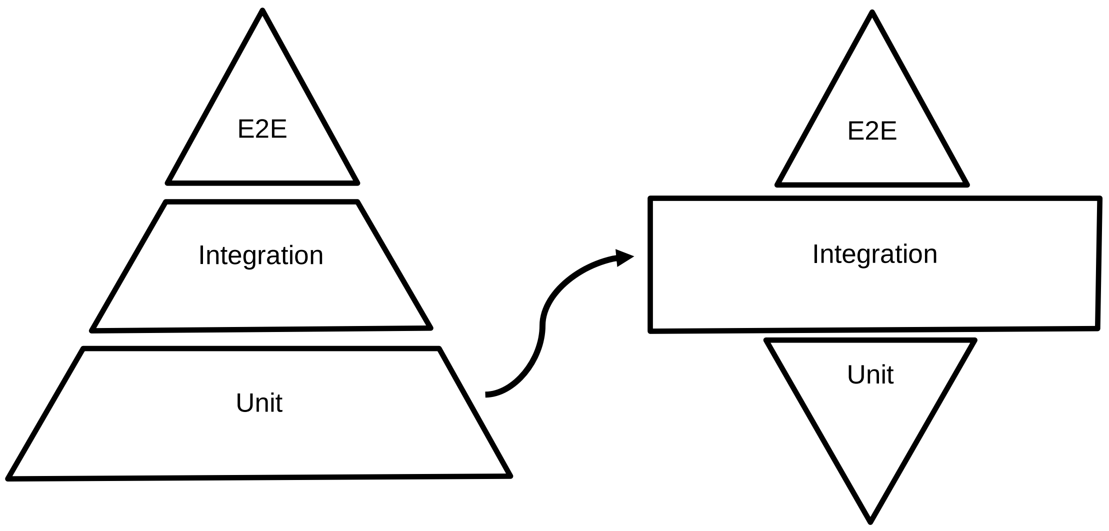
IaC claims to adopt software development techniques, such as version control and automated testing (Morris, 2020, chapter 8; Winkler, 2021). However, there are significant differences between infrastructure and application code:

- Application code can be easily tested on a developer's machine, which is significantly more challenging for infrastructure code. Even though tools like multipass (Canonical, n.d.) allow the configuration of VMs similar to public cloud deployments, this does not apply to all resource types. Brikman (2022) summarized this observation as "you can't use localhost." This statement goes hand in hand with Hummer et al. (2013, p. 386), who emphasized that often IaC must be tested with real infrastructure.
- Unit, integration, and end-to-end testing infrastructure code are slow compared to application code (Morris, 2020, chapter 8).
- Many IaC tools use declarative languages that define the target state, making this code more difficult to test (Morris, 2020, chapter 8). This is because the IaC tool is responsible for applying the changes. Testing whether the target state has been reached would test the IaC tool, not the infrastructure code.

In summary, common testing patterns and practices from application code testing cannot be transferred to infrastructure code. For instance, the often-cited test pyramid (Fowler, 2012) becomes a test diamond, according to Morris (2020). This is because most unit tests are integration tests instead, and because of this, testing effort in the pyramid shifts from the unit testing stage at the bottom to the integration testing stage in the middle. Figure 1 illustrates how different IaC testing is compared to application testing.

To summarize, unit, integration, and end-to-end testing of infrastructure code pose several challenges and require "hard work" (Morris, 2020). This applies to all "online tests" (Morris, 2020, chapter 9) that communicate with infrastructure providers. In contrast, "offline tests" (Morris, 2020, chapter 9) analyze the source code without interacting with infrastructure providers. Static code analyzers check different aspects of the infrastructure code, like security and coding style (Morris, 2020, chapter 9). Improving from no infrastructure tests to a comprehensive test suite, including static code analysis, unit, integration, and end-to-end tests, is a significant effort. This observation suggests starting with **static code analysis**, which poses a valuable beginning because it is fast, license fee-free (if open-source tools are used), stable, reliable, and easy to use (Brikman, 2022, chapter 9).

Figure 1. The Testing Pyramid becomes a Diamond for IaC Testing



Source: Own representation (changed) based on Morris, 2020, chapter 7.

2.4. Static Code Analysis

Static code analysis can be explained best when compared to other testing approaches. Compared to unit, integration, and end-to-end testing, the main characteristic of static code analysis is that the application itself is not executed (Ashfaq et al., 2019). This chapter investigates the state of research and practice of static code analysis tools for general-purpose programming languages like Java. At this point, it is essential to differentiate between general-purpose programming languages and domain-specific languages (DSLs) (Stefanovic et al., 2021). Whereas general-purpose programming languages can be applied to several domains, DSLs are limited to one single domain (Stefanovic et al., 2021). Stefanovic et al. (2021) emphasize that static code analysis for DSLs is more challenging to implement than for general-purpose programming languages.

Zampetti et al. (2017) investigated the usage of static code analysis in 20 open-source Java projects on GitHub. They analyzed how these static code analyzers were used in Travis CI (Travis CI, n.d.) pipelines and why pipelines failed. Zampetti et al. (2017) concluded that most pipeline failures occur because static analyzers detect that the code does not adhere to coding standards. In contrast, pipeline failures due to static analyzers detecting bugs or vulnerabilities appear less frequently. Most often, developers resolve the static code analyzers' findings instead of disabling the specific rules. This observation implies that the code maintainers agree with the findings because if the findings were false positives, they would instead deactivate the particular rule either entirely or only for the specific finding (e.g., by commenting *//NOSONAR* when SonarQube is used for Java projects).

Ashfaq et al. (2019) compared several static code analyzers for Java code. They used nine categories to compare the tools. **Input**, the first category, specifies the input type (e.g., source code or byte code). This category cannot be applied to infrastructure code because infrastructure code always comes as source code. The second category, **rules**, was used to analyze the type of rules each tool implements, like style, concurrency, and exceptions. This category is suitable for the analysis of SICAs, although the rule categories have to be adapted. The rule categorization is highly subjective. Hence it does not allow comparison between different sources. **Extensibility** specifies whether a tool can be extended with custom rules, and **availability** describes whether a tool is free to use. The **reports** were analyzed regarding how parsable they are. Furthermore, the authors analyzed the tools' latest **release date**. **Errors**, the next category, indicates the different types of errors a tool may reveal, for instance, naming convention violations. All these categories, except input, can also be used to analyze SICAs, although the error categories must be adjusted to the given context of infrastructure code. The authors measured the different severity levels the tools used to classify their findings with the category **violation of rules**. The last category, **output**, analyzed the output formats the tools provide for their findings, like XML. Both categories can be used for the SICA assessment too. In summary, only the first category, **input**, cannot be used to analyze SICAs in this work. All other categories are suitable for the SICA assessment in chapter 4.5.

In a similar work, Fatima et al. (2018) compared sixteen tools against twenty-eight measures. Based on the coverage of those measures, the authors ranked the tools. However, these measures cannot be applied to pure declarative infrastructure code. The characteristics, such as infinite loop or unreachable code, mainly apply to application code. Although such constructs are possible for some IaC languages, this does not apply to all. Hence these measures are omitted. Nevertheless, the ranking technique the authors applied can be used for the SICA assessment in this work; however, only for SICAs of the same category. Comparing SICAs for provisioning tools like *Terraform* against SICAs for configuration management tools provides no value since the tools work on very different levels of infrastructure.

Kaur and Nayyar (2020) tested fewer static analysis tools than the sources discussed above (two for JAVA and three for C/C++) against open source repositories. According to the authors, testing tools against open-source repositories provides "standard" results (Kaur & Nayyar, 2020) compared with tests against proprietary repositories. However, it may be valuable to test those tools against both repository types, open-source and proprietary, as will be done in chapter 4.5 of this work. The authors used 118 categories of vulnerabilities from the Juliet test suite (for C/C++) and 171 categories of vulnerabilities from the Apache TOMCAT repository (for Java) as a benchmark. Based on this, they calculated the **detection ratio** for each tool. In addition to the detection ratio, the authors described the **installation effort** for each tool. Their work suggests that using an independent repository with vulnerabilities as a benchmark allows a fair comparison of the tools.

Novak et al. (2010) compared four static analysis tools for application code in ten categories, some of which have already been discussed above (mainly in the discussion of the work of Ashfaq et al. (2019)). The authors analyzed the **release frequency** (frequently, occasionally, obsolete) and the **supported languages**, both of which are suitable characteristics for the SICA assessment in chapter 4.5. All other categories were already discussed above.

Reynolds (2017) outlined that false positives are a common problem in static code analysis. Even though the author referred to application code, the same challenge may occur for infrastructure code. Thus, the **false positive rate** is used in this thesis to assess the SICAs. The authors claimed that high false positive rates are typical. This is because ”it is often better for the tool to state that there is a problem and be wrong (i.e., a false positive) than not to state that there is a problem and be wrong (i.e., a false negative)” (Reynolds, 2017, p. 2).

Most characteristics for assessing static application code analyzers used in formal literature are incorporated into the SICA assessment in chapter 4.5. Obviously, these characteristics have to be adapted to the infrastructure code domain. Whereas this chapter discussed literature on static application code analysis, the following chapter elaborates on **static infrastructure code analysis**.

2.5. Prior Research on Static Infrastructure Code Analysis

A Definition of Static Infrastructure Code Analysis

As explained earlier, static analysis must be considered in the context of the shifting left DevOps and DevSecOps paradigms. Generally, testing can be differentiated into static and dynamic testing (Lewis, 2017; M. Ribeiro, 2022). One major characteristic of static testing is that it is not time-dependent because the program is not executed (Rushgrove, 2020), whereas dynamic testing is (Lewis, 2017). Static analysis only considers the syntax and semantics of the infrastructure code (Chiari et al., 2022). Based on the descriptions of Brikman (2022) and Morris (2020), the following definition is established. Static infrastructure code analysis is performed by tools that automatically analyze the infrastructure code for syntax errors, coding style issues, policy adherence, security flaws, and other misconfigurations without running the code or connecting to a cloud application programming interface (API). In this definition, the term automatically indicates that the analysis is run by a tool, on-demand, or in a continuous integration/continuous delivery (CI/CD) pipeline. This characteristic is decisive because static analysis may also be performed manually via reviews (Lewis, 2017). However, this thesis only considers tool-driven, automated approaches. Lewis (2017) listed 60 testing approaches for **application** testing. In literature, there seems to be no consistent classification of **infrastructure** code testing approaches. Hence, before investigating static infrastructure code analysis in detail, existing classification approaches for infrastructure code testing are examined.

Classification of Infrastructure as Code Testing

Morris (2020) divided infrastructure testing into offline and online testing on the highest level. Offline testing does not require the provisioning of real infrastructure but may include communication with APIs (Morris, 2020). “Syntax checking, offline static code analysis, static code analysis with the platform API, and testing with a mock API” are offline testing approaches, according to Morris (2020). On the other hand, there are several ways to conduct online testing, such as preview, verification, and testing of the outcomes. Table 1 overviews these offline and online testing approaches and provides examples.

Bold entries in tables 1-3 are those testing approaches considered static analysis, whereas crossed-out entries are not.

Table 1. Infrastructure Testing Approaches

#	Name	Description	Offline /Online	Example
I1	**Syntax Checking**	Dry run commands to check the infrastructure code without applying it.	Offline	*terraform validate*
I2	**Offline Static Code Analysis (linting)**	“coding errors, confusing or poor coding style, adherence to code style policy, or potential security issues” (Morris, 2020)	Offline	*tflint*, *cfn-lint*, *cfn_nag*, *tfsec*, *checkov*
I3	~~Static Code Analysis with API~~	Connecting to the cloud API to check for conflicts. It is not static analysis because it involves communication with a cloud API.	Offline	*tflint*
I4	~~Testing with a Mock API~~	Apply the infrastructure code to a mock of the cloud API. It may be used to unit test imperative code. It is not static analysis because the code is executed.	Offline	*localstack*, *Azurite*
I5	~~Preview~~	Preview changes that would be made without applying them. It is not static analysis because it involves communication with a cloud API.	Online	*terraform plan*
I6	~~Verification~~	Check the state and configuration of existing infrastructure. It is not static analysis because it requires real infrastructure.	Online	*awspec*, *clarity*, *InSpec*, *TaskCat*, *terratest*
I7	~~Outcomes~~	Connecting to a server is an example of functional testing (testing the outcomes). It is not static analysis because it requires real infrastructure.	Online	

Source: Own representation based on Morris, 2020.

Apart from this classification of infrastructure code testing, Brikman (2022) differentiated *Terraform* testing into manual and automated testing on the highest level. Table 2 summarizes the different testing approaches.

Table 2. Terraform Testing Approaches

#	**Name**	**Description**	**Manual/ Automated**	**Example**
T1	~~Manual~~	Deploy real infrastructure at best in an isolated sandbox environment. It is not static analysis because it is not automated.	Manual	
T2	~~Unit~~	Test one unit. It is not static analysis because the code is executed.	Automated	Ruby, Go (*terratest*)
T3	~~Integration~~	Test multiple units together. It is not static analysis because the code is executed.	Automated	Ruby, Go (*terratest*)
T4	~~End-to-End~~	Testing the system as a whole. It is not static analysis because the code is executed.	Automated	Selenium (testing “from the end user's perspective” (Brikman, 2022, chapter 9))
T5	**Static analysis**	Parsing and analyzing the code (e.g., syntax, policies).	Automated	*terraform validate*, *tfsec*, *tflint*, *terrascan*
T6	~~Plan testing~~	Analyzing what would be changed by reading, not writing. It is not static analysis because it involves communication with a cloud API.	Automated	*terraform plan*, *terratest*, *Open Policy Agent* (*OPA*), *HashiCorp Sentinel*, *checkov*, *terraform-compliance*
T7	~~Server testing~~	Testing that servers “have been properly configured” (Brikman, 2022). It is not static analysis because it requires real infrastructure.	Automated	*InSpec*, *serverspec*, *goss*

Source: Own representation based on Brikman, 2022.

Lastly, Meijer et al. (2022) differentiated *Ansible* testing tools into linters and verifiers. These approaches are described in Table 3.

Table 3. Ansible Testing Approaches

#	Name	Description	Example
A1	**Linters**	Check syntax, formatting, best practices, and good code style.	*molecule*, *ansible --syntax-check*, *YAMLlint*, *ansible-lint*, *ansible-review*
A2	~~Verifiers~~	Check whether an Ansible role has been successfully run. It is not static analysis because the code is executed	*molecule*, *goss*, *testinfra*, *serverspec*

Source: Own representation based on Meijer et al., 2022.

Even though the described infrastructure code classification approaches are different, the authors do not contradict each other. This is because the three sources have different scopes. Morris (2020) referred to IaC testing in general, whereas Brikman (2022) and Meijer et al. (2022) only considered *Terraform* and *Ansible*, respectively. Since *Terraform* and *Ansible* belong to different IaC categories (provisioning tool and configuration management tool), the described testing approaches suggest that different IaC tool categories may be tested differently. Even though some of the example tools mentioned in tables 1-3 are assigned to one of the testing approaches neglected in this work (e.g., *terratest* is mentioned for unit testing and integration testing), they may still be considered a valid SICA in chapter 4.3. This is because they offer the functionality of several categories, one of which is an appropriate static code analysis category (e.g., *terratest* also offers static code analysis).

Compared to all other testing approaches, static analysis is the easiest, most stable, and fastest to run, although the errors it can detect are weaker (Reynolds, 2017, p. 1). Each entry in Table 1 (entries numbered with *I(nfrastructure)* 1-7), Table 2 (entries numbered with *T(erraform)* 1-7), and Table 3 (entries numbered with *A(nsible)* 1-2) is now classified to be either considered as static analysis or not. I1 (syntax checking) is considered static analysis because the code is not executed. As the name suggests, I2 (offline static code analysis) is also static analysis. In contrast, I3 (static code analysis with API) is not classified as static analysis in this work because it involves communication with a cloud API. Communication with the cloud provider requires interaction over a network, increasing the chance of network and cloud provider-related errors and making the test less reliable and stable. Two main advantages of static infrastructure code analysis are reliability and stability. Also, I4 (testing with a mock API) is not static analysis because the code is actually executed even though only against a mock API. Although I5 (preview) does not run the infrastructure code, it is not pure static analysis because communication with a cloud API is involved (HashiCorp, n.d.–a; Morris, 2020). I6 (verification) and I7 (outcomes) are also not static analysis approaches because those approaches can only be executed when real infrastructure is deployed. Moving on to the *Terraform*-specific testing approaches described by Brikman (2022), only T5 (static analysis) is ranked as static analysis in this work. All other approaches involve executing the code or at least having some infrastructure deployed. From the two *Ansible* testing approaches, A1 (linters) is classified as static analysis, whereas A2 (verifiers) is not because it involves running an *Ansible* playbook. Beyond the testing approaches discussed so far, Hasan et al. (2020, p. 10) analyzed testing practices for IaC and differentiated between linting, behavior testing, remote testing, and sandbox testing, of which only linting is static analysis. The remainder of this chapter describes an IaC misconfiguration, the lack of static infrastructure code analysis-related research, and investigates the rare static IaC testing approaches in formal literature.

Example of an IaC Misconfiguration that can be Identified via Static Analysis

So far, different IaC testing approaches have been explained, and it became apparent that static analysis is the only testing approach considered in this work. Still, how can an IaC misconfiguration appear in infrastructure code, and how can static analysis tools detect it? The following example is adopted from the *checkov* documentation (bridgecrew, n.d.–b). Listing 1 shows a section of a *Terraform* configuration for an *AWS S3* bucket. The access-control list (ACL) is set to *public-read*, which means that the bucket is accessible by anyone. For some use cases, this is the wanted behavior, for example, when static content like a website is hosted. For other use cases, like the storage of sensitive data, this information must only be visible to authorized individuals. Running *checkov* on the *AWS S3 Terraform* configuration yields the output shown in Figure 2.

Listing 1. S3 Bucket Terraform Configuration with Public Access

```
resource "aws_s3_bucket" "foo-bucket" {
  acl           = "public-read"
}
data "aws_caller_identity" "current" {}
```

Source: bridgecrew, n.d.–b.

Figure 2. Checkov Output For a Publicly Accessible S3 Bucket Defined in Terraform

```
Check: "S3 Bucket has an ACL defined which allows public access."
 FAILED for resource: aws_s3_bucket.foo-bucket
 File: /example.tf:1-25

...
```

Source: bridgecrew, n.d.–b.

Checkov assumes by default that ACL buckets should not allow public access and issues a warning. Moreover, it shows information about the exact location of the issue and a description of the problem. The above warning can be ignored for use cases where a publicly accessible S3 bucket is desired. The check can be skipped by adding a comment, as shown in Listing 2.

Listing 2. S3 Bucket Terraform Configuration with Public Access and a Comment to Suppress a Warning

```
resource "aws_s3_bucket" "foo-bucket" {
  #checkov:skip=CKV_AWS_20:The bucket is a public static content host
  acl           = "public-read"
}
data "aws_caller_identity" "current" {}
```

Source: bridgecrew, 2021.

The identification of the misconfiguration is relatively easy in this case because the value of the property *acl* on the resource *aws_s3_bucket* must simply be checked for the value *public_read*, and, in case the value is *public_read*, the check must fail (see bridgecrew (2022) for the code for this check). Albeit, there are also more complicated checks. Adding a comment to the code and running *checkov* again skips the rule *CKV_AWS_20* with the following output (Figure 3).

Figure 3. Checkov Output For a Publicly Accessible S3 Bucket Defined in Terraform with Rule Suppression

```
Check: "S3 Bucket has an ACL defined which allows public access."
 SKIPPED for resource: aws_s3_bucket.foo-bucket
 Suppress comment: The bucket is a public static content host
 File: /example.tf:1-25
```

Source: bridgecrew, 2021.

The reader now has the necessary background and basic knowledge of how static analyzers benefit practitioners and can support the prevention of misconfigurations. Also, it became apparent how essential it is for SICAs to allow the user to skip checks.

Lack of Research on Static Infrastructure Code Analysis

Although IaC tools can be dated back to 1993, when *CFEngine* was introduced (Schwarz et al., 2018), researchers and practitioners criticize that the IaC ecosystem, in general, is still immature (Guerriero et al., 2019; Morris, 2020). Especially tools for IaC testing are highly required (Guerriero et al., 2019). Wilson (2020, p. 158) also claimed that infrastructure code analysis is not as mature as static analysis in application development. In particular, Morris (2020), a leading IaC expert, stated that there are significantly fewer static analysis tools for infrastructure code than for application code, requiring practitioners to investigate the state of practice and potentially close open gaps by developing tools. In the same way, Guerriero et al. (2019) investigated the state of research by conducting interviews with practitioners. Based on their qualitative research, they confirmed the lack of SICAs. Furthermore, by analyzing several papers in the research area of DevOps, Alnafessah et al. (2021) found IaC-related research under-investigated too. Likewise, A. Rahman, Parnin, and Williams (2019) explained the lack of particularly security-related IaC testing research. The observation that only a few research studies on static infrastructure code analysis exist is further underlined by the fact that two of the research studies described below are non-peer-reviewed Bachelor's or Master's theses (Hortlund, 2021; T. J. A. Ribeiro, 2021). A. Rahman (2018a, p. 111) explained that “toolsmiths can use our prediction models to build tools that pinpoint the defective IaC scripts that need to be fixed,” also indicating a lack of such tools. Nevertheless, new tools with similar functionality to existing tools are likely to be developed without a prior investigation of the state of practice. Equally, not only IaC-related research studies recognized a lack of research in this area. Rajapakse et al. (2022) investigated DevSecOps challenges and found limitations of IaC tools and scripts. Furthermore, by analyzing StackOverflow questions on *Puppet* code, A. Rahman, Partho, et al. (2018) found that practitioners ask many questions on syntax errors and security aspects, which, once more, could be targeted by static analyzers.

It is now clear that static IaC testing is an understudied research area. Hence, it is only logical to ask why that is. A. Rahman, Mahdavi-Hezaveh, and Williams (2019) came up with three possible explanations for this question. First, they speculate that IaC could not be widely adopted in the industry. However, the 2022 Stack Overflow Developer Survey (Stack Overflow, 2022) contradicts this assumption. According to the survey, seven of thirteen tools in the category “other tools” are IaC tools. Professional developers use *Docker* (69%), *Kubernetes* (25%), *Terraform* (12%), *Ansible* (10%), *Puppet* (2%), *Chef* (1%), and *Pulumi* (1%) (Stack Overflow, 2022). Interestingly, the three most liked tools in the category “other tools” are IaC tools, too: *Docker* (77%), *Kubernetes* (75%), and *Terraform* (69%) (Stack Overflow, 2022). Second, it is also unlikely that practitioners are unwilling to share their experience with IaC, as shown by the numerous articles investigated in chapters 4.2 and 4.3. Third, what could be a more practical reason for the lack of research in the area is that researchers have limited access to the data because IaC-related repositories may contain sensitive information. This could pose a security threat to companies causing them to keep IaC research and code under lock to avoid security breaches by all means.

Static Infrastructure as Code Testing in Formal Literature

According to bridgecrew (2020b), a 2020-based scan of 2,600 *Terraform* modules with the open-source SICA *checkov* revealed that almost 50% of the *Terraform* modules contained misconfiguration. Even more importantly, this applied to 80% of the most popular modules. Although it remains questionable whether the scan results have been checked manually for false positives because bridgcrew, the backing company behind *checkov*, has an interest in *checkov*’s success in identifying misconfigurations, the result of the scan implies a need for static infrastructure analysis. The reason for the high misconfiguration rate of *Terraform* modules is that many modules use the default values of cloud providers that may speed up development time but, in turn, do not follow best practices (Seymour, 2020). For instance, AWS' managed relational database service (RDS) (Amazon Web Services, n.d.–b) does not enforce transport layer security (TLS) by default (Walikar, 2022, March 11).

Before any static code analyzer can be developed, quality issues like security or development anti-patterns for the particular programming language must be known. This is because static analyzers can only identify known knowns (Schoster, 2020), that is, misconfigurations known beforehand.

Several research studies in formal literature have investigated code smells, bugs, and anti-patterns for infrastructure code. Even though anti-patterns and code smells appear equal, the two terms describe different concepts. An anti-pattern is an approach to a problem that itself has a negative impact (Brown, 1998). For instance, Laigner et al. (2021) describe the concrete class injection anti-pattern for Java in which a concrete class instance is injected. Anti-patterns may be thought of as the opposite of design patterns (Koenig 1998, as cited in Fontana et al., 2016 - 2016, p. 610). On the other hand, code smells are poor software designs that complicate maintainability and may even impact program efficiency or cause bugs in the future (Fowler, 1999; Pereira dos Reis et al., 2022; Yamashita & Moonen, 2013). Security-related issues in IaC scripts are commonly referred to as security smells, defined as “recurring coding patterns that are indicative of security weakness and can potentially lead to security breaches” (A. Rahman, Parnin, & Williams, 2019, p. 164). In one of these studies, A. Rahman (2018a) identified development and security anti-patterns in infrastructure code as well as defect categories. As described before, the work of A. Rahman (2018a) laid the foundation for static code analyzers that scan infrastructure code for the identified anti-patterns.

Moreover, Schwarz et al. (2018) identified 17 *Chef* code smells. They also established a novel classification of code smells. Their research shows that code smells can be distinguished between technology-agnostic, technology-dependent, and technology-specific. Furthermore, the authors discovered a lack of general IaC catalogs. The fact that static code analyzers are language-specific (Bhuiyan & Rahman, 2020) encourages the existence of technology-dependent smells. Schwarz et al. (2018) furthermore specified two essential quality characteristics of static infrastructure code analyzers: soundness and completeness. According to the authors, those characteristics exist in a natural trade-off. The work of Schwarz et al. (2018) gives essential input for developing a static *Chef* analyzer in much the same way the work of A. Rahman (2018a) lays the foundation for building a static analyzer for *Ansible*. However, since both sources were created four years ago and *Ansible* and *Chef* were constantly updated in the meantime, the results must be checked for validity.

In addition to these older research studies on static infrastructure code analysis, Chiari et al. (2022) recently analyzed the state of research on static infrastructure code analysis. This is the latest study available. In their work, the authors investigated existing tools, how they work, and which IaC tools they support. Chiari et al. (2022) explained that their results could help researchers guide future research. Nevertheless, they only considered formal literature, thereby neglecting a considerable number of SICAs for which no research study exists. Also, the authors only considered *Ansible*, *Puppet*, *Chef*, *Terraform*, and *Cloudify* in their search string as IaC tools, omitting many other popular IaC tools, as will be shown in chapter 4.2. They collected 42 categories of code smells that can be used to evaluate IaC tools, although some only apply to configuration management tools like *Ansible*. The work of Chiari et al. (2022) was published just before this work was started. Ultimately, before researchers start a new research project based on the findings of Chiari et al. (2022), they may consider the result of the present work and check whether practitioners have already covered their research project. Chiari et al. (2022, p. 218) identified several “SICA properties.” These properties can also be applied to the assessment of SICAs in chapter 4.5. SICAs can have the following target properties: detect anti-patterns, code smells, or security vulnerabilities (Chiari et al., 2022, p. 224). Furthermore, Chiari et al. (2022) introduced the target properties of deployment plan correctness, determinism, idempotency, network policies, bugs in shell scripts, undefined references, linear temporal logic (LTL) on deployed services behavior, and success of orchestrated operations. Nevertheless, these target properties are specific to different types of IaC tools.

Special attention must be given to the work of Luzar et al. (2021) and the PIACERE project (Tecnalia, n.d.) in general. Luzar et al. (2021) investigated several SICAs specialized in security analysis. The authors referred to those tools as static application security testing (SAST) tools. They also described several meta-linting tools that combine various other SAST tools. The authors used this knowledge to build a meta-tool called *IaC Scan Runner* with Python. The tool combines 21 other SAST tools. Luzar et al. (2021) is the only prior work that mentions and uses several SICAs developed by practitioners. Even so, the authors focused on security analyzers only. None of the mentioned SICAs was assessed, and no prior work has examined the state of practice of SICAs.

In contrast to the research studies mentioned above, which all focus on specific IaC tools (e.g., *Ansible*), in another work, A. Rahman et al. (2021) described practices to handle sensitive information in infrastructure code. Their work exposes the need for tools to identify these so-called hard-coded secrets automatically. If developers do not detect hard-coded secrets, those secrets may be found by bad actors, for instance, in open-source repositories to conduct malicious actions. Beyond that, M. R. Rahman et al. (2022) specialized in a self-developed secret detection tool and analyzed its usage via a case study. The authors found that even though the findings of the secret scanner were accurate, developers tended to bypass it. However, such secret scanning tools do not work on an IaC tool-specific basis but must scan entire infrastructure code repositories. Since these tools are not specific to IaC, they are not investigated in this work.

All the research studies discussed so far examined misconfigurations in IaC tools which is the basis for static analyzers. However, none of these studies developed such a tool. Research studies that designed and developed static infrastructure code analyzers are discussed now. In total, four prior research studies have worked on the creation of static infrastructure code analysis tools. Shambaugh et al. (2015) developed a static code analyzer for *Puppet* (Puppet, n.d.) in Scala. The tool checks whether *Puppet* infrastructure code is deterministic and idempotent. According to the authors, this tool, *Rehearsal*, may be extended in the future to perform security scanning. Because the last change was done almost three years ago, it would have to be evaluated whether the tool works with newer *Puppet* versions. In addition to *Rehearsal*, A. Rahman (2018a) developed the *Security Linter for Infrastructure as Code* (*SLIC*) for *Puppet* (Progress Software Corporation, n.d.) using a parser and a rule engine explicitly scanning for security misconfigurations. In the author's own replication study, a similar tool, the *Security Linter for Ansible and Chef scripts* (*SLAC*), was created to analyze *Ansible* and *Chef* infrastructure manifests (A. Rahman, Rahman, et al., 2019). In the first step, the authors identified security issues in *Ansible* and *Chef* infrastructure manifests. The result of this analysis was then incorporated into the second step, in which the *SLAC* was developed. Hortlund (2021) used the *SLIC* to assess security smells in open-source repositories in another replication study of A. Rahman (2018a). The author found that, on average, *Puppet* manifests contain slightly more security smells than *Ansible* manifests. T. J. A. Ribeiro (2021) also analyzed the *SLIC* and built a new tool in Ruby, *puppet-sec-lint*, to address some of the weaknesses of the *SLIC*. T. J. A. Ribeiro (2021) claimed about the *SLIC* that "although the author [A. Rahman (2018a)] provided very impressive results in terms of precision of his tool, later independent studies revealed its countless flaws that caused an incredibly high number of false positives" (T. J. A. Ribeiro, 2021, p. 51). This statement underlines that although some static infrastructure code analyzers exist and have been developed in formal literature, it does not mean they are suitable for practical usage. In addition to the development of *puppet-sec-lint,* T. J. A. Ribeiro (2021) developed a Visual Studio Code (VS Code) extension for the tool, which is one way to improve practical usage and integration into the development workflow. Beyond that, T. J. A. Ribeiro (2021, p. 51) explained that he considered usability right from the start, emphasizing that this aspect is important for similar research studies in the future. Also, the author designed *puppet-sec-lint* to be easily extendable for natural language processing (NLP) or machine learning (ML) approaches (T. J. A. Ribeiro, 2021, p. 52). Dalla Palma et al. (2020) developed a static analysis tool for *Ansible* in Python, which can be installed as a VS Code extension too. The tool, *AnsibleMetrics*, analyzes 46 metrics in *Ansible* scripts, including bad practices but also more basic metrics like the number of blank lines (Dalla Palma et al., 2020). According to the authors, the tool can easily be adjusted to analyze other IaC tools, like *Puppet*.

All previously discussed research studies created new static code analysis tools. Still, this does not imply that all research studies must create a tool from scratch. In fact, A. Rahman (2018b) described that he plans to incorporate his findings into *ansible-lint* (Thames, 2014), *puppet-lint* (Sharpe, 2022), and similar tools. Moreover, *puppet-lint* was extended by van der Bent et al. (2018) to generate a set of *Puppet* quality metrics like file length and complexity. In contrast to the infrastructure code of configuration management tools, Schermann et al. (2018) analyzed over 100,000 *Dockerfiles* on GitHub with *hadolint* (hadolint, 2015). They made their dataset with the findings of *hadolint* publicly available. The authors did not perform any further analysis on the dataset but left this open to other researchers.

To this point of the chapter, only static infrastructure code analyzers for specific IaC tools (e.g., *Ansible* or *Docker*) have been discussed. In contrast to the technology-agnostic approaches discussed, Saavedra and Ferreira (2022) developed a static infrastructure code analyzer that transforms infrastructure manifests into an "intermediate representation" (Saavedra & Ferreira, 2022, p. 1). These infrastructure manifests' representations are then used to perform security analysis. *GLITCH*, the tool developed by Saavedra and Ferreira (2022), currently supports nine security smells for *Ansible*, *Chef*, and *Puppet*. The main advantage of *GLITCH* compared to other SICAs is that there is only one analyzer for all IaC tools. Therefore, new rules must only be implemented once instead of in every technology-agnostic SICA (Saavedra & Ferreira, 2022). Although the motivation for such a polyglot defect smell detection tool is promising, it is questionable whether the evaluation of *GLITCH* is meaningful. The authors compared their tool with *SLIC* and *SLAC*, which, according to the authors, are two state-of-the-art SICAs. Yet, as discussed earlier, these tools have "countless flaws" (T. J. A. Ribeiro, 2021, p. 51). Simply put, *GLITCH* unites *SLIC* and *SLAC* but only adds two additional checks and little functionality. A further problem is that *GLITCH* adopts the existing flaws of *SLIC* and *SLAC*. Also, it is questionable whether *SLIC* and *SLAC* can be denoted state-of-the-art because the practical adoption of both tools seems to be very low according to GitHub statistics[2]: *SLIC-Ansible* has five stars on GitHub (A. Rahman, 2019), whereas *ansible-lint* has over 2,800 (Thames, 2014). Amid all criticism, polyglot security smell detection tools are promising. However, finding a reason to justify developing such a tool from scratch could be challenging. This is because there are already many popular SICAs; for example, *checkov*, *KICS*, and *terrascan* support several IaC tools. Admittedly, these SICAs do not support multiple configuration management tools like *GLITCH*.

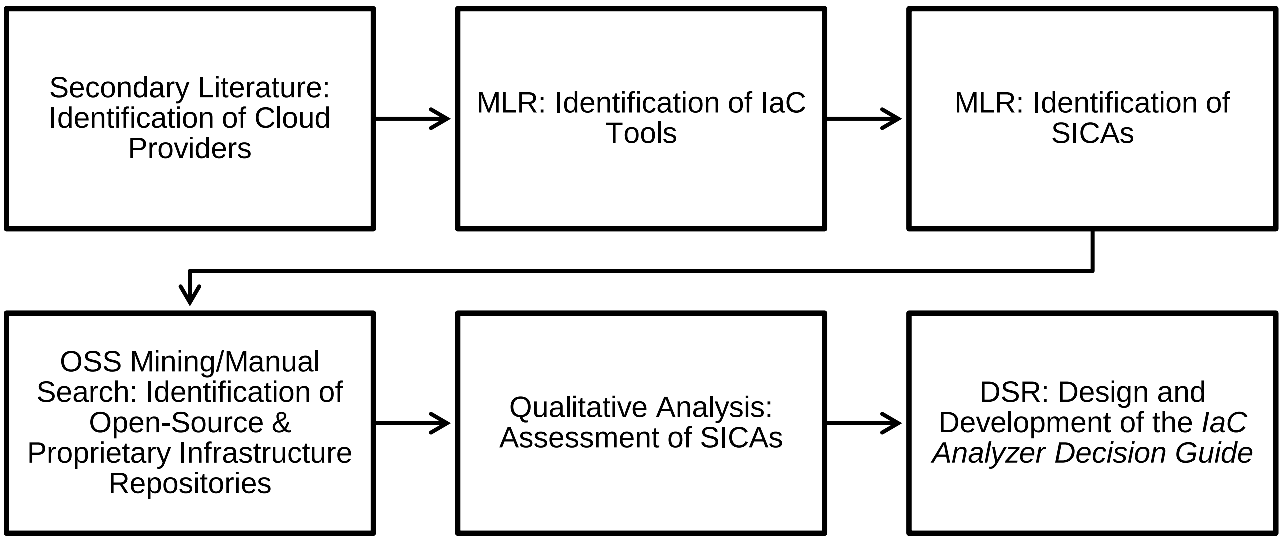
Beyond the sources mentioned before, there are no other research studies in formal literature at the time of writing. The described research studies are the basis for further research on static code analysis in formal literature. However, practitioners often develop SICAs based on industry best practices that are not described in the formal literature. Hence, two parallel and isolated knowledge bases exist – formal literature and informal literature. This isolation is comparable to the development and operation silos in DevOps. Only one research study has aimed to overcome those silos (Luzar et al., 2021). This study is neither peer-reviewed nor published in a popular journal or conference. This may explain why it has not been considered in formal literature yet. Thus, the described research studies cover only a tiny proportion of the available knowledge about quality issues and static analyzers in infrastructure code. Another shortcoming of the described formal research articles is they often only consider configuration management tools (*Ansible*, *Puppet*, and *Chef*). All these observations emphasize the need to bridge the gap between research and practice. With this need in mind, the following chapter describes the **methodology** for bridging this gap.

[2] The provided GitHub statistics represent the state at the time of writing. This information will likely change, which also applies to all other GitHub statistics in this thesis.

3. Research Methodology

This chapter elaborates on the research methodology of this work. It is divided into six parts, starting with identifying the most popular cloud providers and ending with designing and developing the *IaC Analyzer Decision Guide* for practitioners. The result of each chapter is an input for the following chapters. Figure 4 illustrates the six phases.

Figure 4. This Work Follows Six Consecutive Phases


Secondary Literature: Identification of Cloud Providers
MLR: Identification of IaC Tools
MLR: Identification of SICAs
OSS Mining/Manual Search: Identification of Open-Source & Proprietary Infrastructure Repositories
Qualitative Analysis: Assessment of SICAs
DSR: Design and Development of the IaC Analyzer Decision Guide

Source: Own representation.

Before describing the research methodology for each chapter in-depth, general information about the multivocal literature review methodology is presented because it is used in chapters 3.2 and 3.3.

Multivocal Literature Reviews

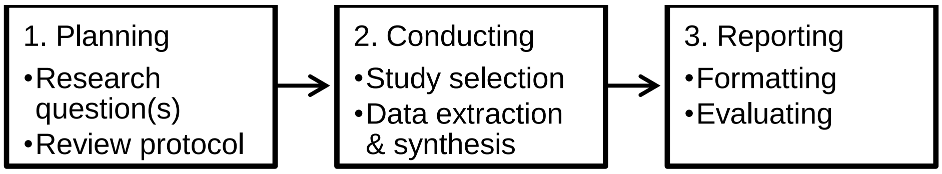
The state-of-the-art IaC tools and the corresponding static analyzers are analyzed using **multivocal literature reviews (MLR)**, thereby conducting **secondary research** by aggregating the knowledge of multiple primary studies (Garousi et al., 2019, p. 5)**.** MLRs are suitable for investigating the state-of-the-art and -practice from a practitioner's and researcher's perspective in software engineering (Garousi et al., 2019). Since this work targets researchers and practitioners alike, MLRs are a suitable choice. In comparison to systematic literature reviews (SLRs), MLRs also investigate a large amount of grey literature (GL) (e.g., blog posts or white papers) (Garousi et al., 2019). In other words, "as SE is a practitioner-oriented … field" (Garousi et al., 2019), practitioners' literature should be considered too. Also, grey literature offers an opportunity to include experience-based knowledge. In summary, the "use of multivocal literature reviews is in closing the gap between academic research and professional practice" (Elmore, 1991). This statement strongly suggests that MLRs are exceptionally suitable for this work. Conversely, an MLR aspires to benefit researchers and practitioners alike (Garousi et al., 2019). Interestingly, "practitioner interviews done and reported by researchers have, for long, been considered as academic evidence in empirical software engineering, while grey literature produced by the very same individuals may have been ignored as unscientific" (Garousi et al., 2019, p. 28). The present thesis overcomes this problem by incorporating grey literature. Seeing that, an explanation of grey literature is required. According to Higgins and Green (2008, p. 106), GL is "literature that is not formally published in sources such as books or journal articles." Garousi et al. (2019, p. 4) describe three types of grey literature, which differ in the level of outlet control and expertise. For instance, whereas the expertise is known for books, this does not apply to short messages on the social media platform Twitter. There is no doubt that the differentiation and classification of grey literature are essential for MLRs where the actual quality of the content of the sources is relevant. Even so, in this work, the grey literature sources are only used to extract IaC and SICA tool names, which the author later manually filters. Hence, grey literature of poor quality may introduce tools that are not IaC tools or SICAs, but this does not influence the quality of the findings. Furthermore, Garousi et al. (2019, p. 4) explain the importance of investigating the producer of GL. In the context of this work, commercial tool vendors may publish blog posts that advertise their offerings, thereby distorting the overall popularity of tools.

The main reason the MLR methodology is selected to identify IaC tools and SICAs is that there is no other feasible way to create a list of these tools. This is because no other primary research has been conducted in this field. Moreover, there is no database listing all such tools. The inclusion of GL into the literature review is subject to the same procedure as formal literature in formal literature reviews. For instance, the Google search results for the MLRs are restricted to the top 100. However, the search is continued if the 100th result still provides valuable content. This procedure is a mixture of effort bounded and evidence exhaustion stopping criteria and is adopted from Garousi and Mäntylä (2016) (mentioned in Garousi et al. (2019)). In addition to the grey literature found via a search engine, formal literature is searched with the same search strings in a formal database. Then, forward and backward snowballing (if necessary) further enlarges the number of identified relevant formal literature sources. The guidelines of Wohlin (2014) are applied to snowballing. As with the Google search, the main exclusion criterion for formal research studies is if articles were published before 2020. This is because the IaC tool ecosystem is constantly evolving (Brikman, 2022, chapter 8). Thus older articles may be outdated. The MLRs in this work follow the three typical phases (Garousi et al., 2019):

1. Planning the review (chapter 3.2, chapter 3.3)
2. Conducting the review (chapter 4.2, chapter 4.3)
3. Reporting the review (chapter 4.2, chapter 4.3)

The same three phases are usually also executed in SLRs. Figure 5 illustrates an adapted version of the MLR process in SE.

Figure 5. An Overview of the Typical MLR Steps



Source: Own representation based on Kitchenham & Charters, 2007, p. 6.

This work uses an adapted, simplified version of the MLR guidelines of Garousi et al. (2019) because their guidelines are for scientific research studies whose main deliverables are the MLRs. In this work, in contrast, the results of the MLRs are only a pre-condition to the assessment of SICAs and the development of the *IaC Analyzer Decision Guide*, which are the main deliverables. Thus, more time shall be invested in assessing the SICAs than conducting the MLRs. The three phases of an MLR, as outlined by Garousi et al. (2019), are now examined in more detail, and, in case the described methodology is adjusted, the reasons are outlined. Indeed, Garousi et al. (2019, p. 12) explicitly stated that their guidelines could be seen as a template that can be extended and adapted by other researchers.

1. The first step, planning the review, consists of establishing the need for the MLR and **establishing the goal and research questions**. The need for MLRs has already been explained in the literature review in chapter 2. Hence, only the goal and research questions are specified. The RQ(s) drive the entire MLR and are, therefore, an important aspect. They must benefit the specified target audience, which, in this work, comprises researchers and practitioners.
2. After the review has been planned, it is conducted in several steps. First, the **search strings** are defined, which might be an iterative process (Garousi et al., 2019, p. 16). Equally, the **search databases** and **search engines** must be selected (Garousi et al., 2019, p. 15). **Stopping criteria** must be chosen to set clear boundaries for the review (Garousi et al., 2019, p. 16). Those can be theoretical saturation (no new insights), effort bounded (stop after N hits), or evidence exhaustion (consider all sources) (Garousi et al., 2019, p. 16). **Selection criteria** and **quality assessment criteria** must be constructed in the next step. Regardless, only a few quality assessment criteria (e.g., date) are applied in this work because, as mentioned earlier, the MLRs are used to establish a list of tools based on the identified articles. The primary qualitative assessment of the identified tools is performed in a subsequent step. Hence, not the sources themselves (restricted to very minimal criteria) but the identified tools are subject to quality assessment. This quality assessment step is shifted from the study selection into the data synthesis. Next, **data extraction** from the sources must be performed manually since the tool names are unknown beforehand. Because no mapping is involved in the MLRs in this work, data extraction is straightforward: Scanning the text and extracting all mentioned tool categories and tools. The extracted data is mainly quantitative in that only the tool and category names are investigated, not their context. Two different types of **data synthesis** are performed for the two RQs per MLR. Quantitative synthesis is performed to collect tools (i.e., answers the question of which tools exist and how many sources mention them). Beyond that, thematic analysis is performed to collect categories of tools (i.e., answers the question of which categories are used to classify the tools) (Garousi et al., 2019, p. 24).

3. Last, the findings are **reported**. The report of the results must be valuable for the entire target group, that is, practitioners and researchers (Garousi et al., 2019, p. 26). To an extent, the *IaC Analyzer Decision Guide* developed in chapter 4.6 can be seen as a report suitable for the entire target group of practitioners and researchers. Garousi et al. (2019, 26p) explained that MLRs might also have different group-specific reports. In addition to the *IaC Analyzer Decision Guide*, this thesis, especially chapter 4.5, is the report for researchers. The authors of the MLR guideline have created a checklist for practitioners, which is very similar to the *IaC Analyzer Decision Guide* developed in this work. Also, the authors make their Google Doc spreadsheet publicly available so that it can be constantly updated, which also applies to the *IaC Analyzer Decision Guide* (Garousi et al., 2019, p. 27).

Now all steps of a multivocal literature review have been explained and adapted to the context of this work. Next, the methodologies of this work's six phases are explained. To begin with, the methodology to identify public cloud providers is elaborated on in chapter 3.1. Then, in chapters 3.2 and 3.3, the MLRs to identify IaC tools and SICAs are specified in more detail. Chapter 3.4 explains how open-source and proprietary repositories to test the SICAs are found, and chapter 3.5 describes the procedure to assess the identified SICAs. Finally, chapter 3.6 outlines the design science research process for developing the *IaC Analyzer Decision Guide*.

3.1. Identification of Public Cloud Providers

Before investigating existing IaC tools, it must be analyzed which public cloud providers these tools use to create, update, or delete resources in the cloud. Public cloud providers offer IT resources like VMs and services to anyone with a credit card (Brassel & Gadatsch, 2020; Li et al., 2011). On the one hand, some IaC tools are cloud provider-specific. This is because several cloud providers offer their own IaC tools, for instance, *AWS CloudFormation* (Amazon Web Services, n.d.–d). On the other hand, some IaC tools support multiple cloud providers. Many cloud providers offer modules for these IaC tools (e.g., the *terraform-aws-provider* for *Terraform*).

According to Turbonomic (2021, March 2), in 2021, 34% of all companies relied on the private cloud. This finding suggests elaborating on the private cloud platforms as well. However, this work is **limited to public cloud providers** because IaC tools and SICAs for private cloud platforms are likely to be proprietary, and only tools that are accessible to the reader free of charge are included. Still, IaC tools suitable for the public and the private cloud, like *Terraform*, are included in this work. Furthermore, only the category of provisioning tools can be specific to a particular cloud provider. All other IaC tool categories work across cloud providers. Beyond this limitation, this work considers the public cloud providers most companies use since IaC tools for those providers are the most interesting for most practitioners and researchers. Then, in chapter 3.2, only IaC tools for those public cloud providers are considered. For instance, if AWS were not one of the most popular cloud providers, *CloudFormation* would not be considered in this work as an IaC tool.

Some SICAs may be specific to the IaC tool of a public cloud provider. Thus, in chapter 4.3, cloud provider-specific SICAs (e.g., *cfn-lint* for *CloudFormation* (AWS Cloudformation, 2018)) are only selected for the identified public cloud providers.

The applied methodology to create such a list of public cloud providers is simple. Cloud computing is subject to constant research and market analysis because it is a prevailing paradigm in the industry. Thus, a report with the most popular public cloud providers is analyzed (secondary research). In contrast, primary research in the form of interviews or questionnaires could not deliver results generally valid for most readers of this work, that is, valid across countries and industries. In essence, the following research question guides this chapter: **RQ 3.1.1**: *What are the top five most popular public cloud providers?*

3.2. Identification of IaC Tools

This chapter identifies existing IaC tools. The result of the previous chapter, the list of the most popular public cloud providers, is used to identify provisioning tools. In addition, all other public cloud provider-independent categories of IaC tools are identified. The general procedure of MLRs has already been described at the beginning of chapter 3. Hence, it is now refined for the specific use case of IaC tools.

3.2.1. Planning the Review

Various IaC tools exist, and each of them may come with built-in static analysis support and several additional open-source or paid static analyzers. This chapter chooses the most popular among the IaC tools in each category (i.e., the samples), and chapter 3.3 dives deeper into static IaC analyzers for those tools. This is because a definitive analysis of all static IaC analyzers for all IaC tools is impossible due to the scope of this work. To make matters worse, the entire population of IaC tools is inaccessible (Thomas, 2021, p. 139). That is, there is no list of IaC tools. The samples are chosen according to a scoring method, selecting only the most popular tools. The following research questions guide this multivocal literature review:

- **RQ 3.2.1**: *How do practitioners and researchers categorize IaC tools?*
- **RQ 3.2.2**: *What IaC tools exist?*

RQ 3.2.1 aims to find alternative categorization approaches in addition to the already known ones from formal literature and books (e.g., Morris (2020)) to add new search strings and thereby identify more IaC tools. For instance, Morris (2020) defines *Helm* as a deployment tool, which qualifies it for further evaluation. Similar deployment tools could be searched if *Helm* met the criteria to be considered an IaC tool in this work.

3.2.2. Conducting the Review

The steps to selecting suitable IaC tools are:

1. Extraction of IaC tools from formal literature.

2. Extraction of IaC tools from grey literature.
3. Data extraction and synthesis.

Extraction of IaC Tools from Formal Literature

First, the following search terms are used in the Ebsco database (https://www.ebsco.com): *Infrastructure as code tools*, *configuration management tools*, *server templating tools*, *orchestration tools*, *provisioning tools*, and *cloud infrastructure automation*. The inclusion and exclusion criteria shown in Table 4 are used for the formal literature review.

Table 4. Inclusion and Exclusion Criteria for the Formal Literature Review on IaC Tools

#	**Criterion**	**Inclusion criterion**	**Exclusion criterion**
1	Date	Published 2020-2022	Published before 2020
2	Availability	Full text available	Restricted access

Source: Own representation.

Extraction of IaC Tools from Grey Literature

Second, the systematic grey literature review is used to identify IaC tools. The grey literature review is oriented towards Kumara et al. (2021), who used a similar research approach by analyzing blog posts and white papers. The criteria listed in Table 5 are used to filter the search results in a Google Chrome Incognito Tab to minimize the bias introduced by cookies (Blakeman, 2018; Ćurković & Košec, 2018). The same search strings as in the formal literature search are used during the search process.

Table 5. Inclusion and Exclusion Criteria for the Grey Literature Review on IaC Tools

#	**Criterion**	**Inclusion criterion**	**Exclusion criterion**
1	Date	Published 2020-2022	Published before 2020
2	Result Category	A blog post or similar	Ad, Company related to a proprietary IaC tool that only mentions its own tools.
3	Content	Does mention IaC tools	Does not contain IaC tools

Source: Own representation.

Initially, it was planned to include verbatim results only, as explained by Ćurković and Košec (2018). However, this is impossible in the Google search engine when simultaneously filtering for a specific time range. Furthermore, due to the high number of results, stopping criteria must be specified. For this reason, it is determined to select only the top 100 results for further analysis for each search term. If the 100th result provides valuable information, the search is continued.

3.2.3. Data Extraction and Synthesis

Third, after extracting IaC tools from formal and informal sources, the list of identified IaC tools must be filtered from non-IaC tools. That is, the list is subject to qualitative assessment. Non-IaC tools, in this context, are mentioned in one or more of the analyzed sources but do not meet the criteria defined below.

In the first iteration of the qualitative assessment process, the documentation of each identified tool is quickly scanned to pre-filter apparent misfits. For instance, it may be self-evident if a tool is commercial. Those tools are then excluded without additional assessment. In the second iteration, the remaining tools are evaluated based on the criteria outlined in Table 6. Only tools that meet all those criteria are considered IaC tools and are used in the remainder of this work.

Table 6. Inclusion and Exclusion Criteria for the Identified IaC Tools

#	**Criterion**	**Inclusion criterion**	**Exclusion criterion**
1	Functionality	Used to define, create, update, or destroy infrastructure automatically.	Used to do similar infrastructure-related tasks.
2	Infrastructure Definition	Defined as Code	Configured via UI
3	Usage	Apply code via a tool	Command-line interface (CLI), API
4	Availability	Free[3]	Commercial[4]
5	State of Development	The last commit on the main/master branch is not older than one year. The start date of this thesis is selected as the threshold (May 11, 2022).	The last commit is older than one year, or the project is archived.
6	Supported Cloud Providers[5]	Supports one or more of the cloud providers identified in the previous chapter.	Supports none of the cloud providers identified in the previous chapter.

Source: Own representation.

Based on the answers to **RQ 3.2.1** (categories of IaC tools) and **RQ 3.2.2** (most popular IaC tools), the four most popular tools of each category (i.e., the samples) are selected for further analysis, given more than five sources mention them. Following the selection, static infrastructure code analyzers for these IaC tools are identified.

3.3. Identification of Static Infrastructure Code Analyzers

Similar to the identification of IaC tools, this chapter describes the procedure to identify static analyzers for the previously explored IaC tools.

[3] Some tools come with a Freemium model. These tools can be used for free but additional paid features are available. For those tools, the criterion is still met if the basic functionality is available for free.
[4] Commercial tools are excluded to restrict the results to IaC tools that are accessible to everyone.
[5] Applies only to provisioning tools.

3.3.1. Planning the Review

Many of the IaC tools identified with the previously described procedure come with built-in linting and validation tools (e.g., *Helm* and *Terraform,* which are already known from the literature review). These tools often support standard syntax checks (Brikman, 2022). However, for more advanced industry-standard checks, additional tools exist. On the one hand, at the time of writing, no formal literature study has analyzed the state-of-the-art and -practice of existing SICAs beyond formal literature. Even though Luzar et al. (2021) and, more generally speaking, the PIACERE project mention multiple SICAs, there is no extensive list yet. On the other hand, practitioners created various articles summarizing some of the most popular SICAs. This observation clearly shows the need for a multivocal literature review to establish a comprehensive list of SICAs. The following research questions guide this multivocal literature review:

- **RQ 3.3.1**: *How do practitioners and researchers categorize SICAs?*
- **RQ 3.3.2**: *What SICAs exist?*

RQ 3.3.1 aims to find categorization approaches that suggest decisive characteristics of the SICAs. These characteristics can then be used in chapter 4.5 to describe and categorize the identified SICAs.

3.3.2. Conducting the Review

The steps to selecting suitable SICAs for this work are:

1. Extraction of SICAs from formal literature.
2. Extraction of SICAs from grey literature.
3. Data extraction and synthesis.

Extraction of SICAs from Formal Literature

First, formal literature is searched via the Ebsco (https://www.ebsco.com) database. The search strings used to identify SICAs are *Infrastructure as Code* ^ *Static Analysis*, *Infrastructure as Code* ^ *Security Scanning*, *Infrastructure as Code* ^ *Security*, and *Infrastructure as Code* ^ *Linting*. The inclusion and exclusion criteria shown in Table 7 are used for the formal literature review.

Table 7. Inclusion and Exclusion Criteria for the Formal Literature Review on SICAs

#	**Criterion**	**Inclusion criterion**	**Exclusion criterion**
1	Date	Published 2020-2022	Published before 2020
2	Availability	Full text available	Restricted access

Source: Own representation.

Extraction of SICAs from Grey Literature

Second, the search strings from the formal literature review are also applied to the informal literature review. This informal literature review is subject to the following inclusion and exclusion criteria, as in the informal literature review for IaC tools. The criteria are summarized in Table 8.

Table 8. Inclusion and Exclusion Criteria for the Grey Literature Review on SICAs

#	**Criterion**	**Inclusion criterion**	**Exclusion criterion**
1	Date	Published 2020-2022	Published before 2020
2	Result Category	A blog post or similar	Ad
3	Content	It does mention static **infrastructure** analyzers.	It does not mention SICAs.

Source: Own representation.

3.3.3. Data Extraction and Synthesis

Third, after the literature reviews, the identified tools are filtered with the criteria outlined in Table 9 so that only tools that meet the criteria are included.

Table 9. Inclusion and Exclusion Criteria for the Identified SICAs

#	**Criterion**	**Inclusion criterion**	**Exclusion criterion**
1	Functionality	Static code analysis	Dynamic testing or similar
2	Scope	Infrastructure code	Application code
3	IaC	Specific to IaC	Syntax checking or formatting for all JSON, YAML, or other files (e.g., YAMLlint (Vergé, 2016)).
4	Availability	Free[6]	Commercial
5	State of Development	The last commit on the main/master branch is not older than one year. The start date of this thesis is selected as the threshold (May 11, 2022).	The last commit is older than one year, or the project is archived.

Source: Own representation.

Criterion #1 has been applied by Chiari et al. (2022) as well, who performed a similar analysis of SICAs considering only formal literature. Because of criterion #1, container scanning tools are not considered SICAs even though they perform static analysis. However, they do not scan the IaC manifests (the *Dockerfile*) themself. In contrast, a tool scanning the *Dockerfile* itself is considered a SICA.

If more than ten SICAs meet the inclusion criteria and have more than five votes, a scoring method is applied so that only the top ten SICAs are assessed in detail. This scoring method is based on the

[6] Some tools come with a Freemium model. These tools can be used for free but additional paid features are available. For those tools, the criterion is still met if the basic functionality is available for free.

popularity of the tools. The popularity is measured by the number of votes from the analyzed source. The top ten SICAs with five or more votes are selected for the in-depth assessment. It could be that some of the selected IaC tools are not covered. In this case, the top SICA for each IaC tool is also selected for the in-depth analysis, although having less than five stars. In exchange, the least popular SICA for an IaC tool already covered by another SICA is removed. Thereby, only ten SICAs are selected for the in-depth assessment. To put it briefly, all identified SICAs that meet the inclusion criteria are analyzed. However, only ten are used for the in-depth assessment.

With the methods described so far, public cloud providers, IaC tools, and static analyzers for these IaC tools have been identified. Thereupon, open-source and proprietary repositories must be identified. Ultimately, these repositories are used to test the selected SICAs.

3.4. Identification of Open-Source and Proprietary Infrastructure Repositories

3.4.1. Open-Source Repositories

With the methodology of chapter 3.3, static infrastructure code analyzers that meet this work's requirements have been collected. Those identified SICAs are subject to qualitative analysis, which is explained in more detail in chapter 3.5. The SICAs must be executed against open-source and proprietary repositories that contain IaC manifests of the particular IaC tools for the in-depth assessment of the SICAs. Since an in-depth analysis of all identified SICAs is out of scope, only the top-scored SICAs are tested as described above. The GitHub GraphQL API (GitHub, n.d.) is queried to find suitable open-source repositories for the identified SICAs. The GitLab API would have to be queried, too, to achieve a more generalizable result. Still, in this work, only GitHub is used for simplicity. The query results are then filtered according to the criteria defined in Table 10. Random sampling (Thomas, 2021, p. 137) is used to choose samples among the identified open-source repositories. Some repositories are identified using the RADON tools developed by Dalla Palma et al. (2021), which can be used to crawl for repositories and collect repository metrics. The repository metrics are essential to deciding on suitable repositories. The reason is that the selected repositories must be as representative as possible to allow generalization (Thomas, 2021, p. 136). However, not all repositories can be identified with the RADON tools. This is because GitHub does not consider each IaC tool's DSL a unique programming language. For instance, RADON can be configured to search for repositories with HCL (the *Terraform* DSL) as the primary repository language. However, this does not work for Ansible, which uses YAML. Filtering for YAML repositories would yield many non-*Ansible* results. Hence, the author must identify many repositories via the GitHub UI. The inclusion and exclusion criteria described in Table 10 are applied to the main branches of the identified repositories (like in A. Rahman, Parnin, & Williams, 2019). Notably, A. Rahman, Parnin, and Williams (2019) explained that although searching open-source repositories may identify training repositories, security smells in those repositories must be considered. This is because beginners often adopt these misconfigurations as they learn to apply the IaC tool. To sum it up, the following research question guides this chapter: **RQ 3.4.1**: *For each selected IaC tool, what are two suitable open-source repositories to conduct static code analysis?*

Table 10. Inclusion and Exclusion Criteria for the Selection of Open-Source Repositories

#	**Criterion name**	**Inclusion criterion**	**Exclusion criterion**
1	Availability	Open-source	Closed source
2	Commits per Month (on average)	≥ 2	< 2
3	IaC Proportion	≥ 11%	< 11%
4	GitHub Stars	≥ 140	< 140
5	Clone	Is not a clone	Is a clone
6	Contributors	≥ 6	< 6
7	License	Specified, according to the OSI definition of open-source.	Not specified or not in line with the OSI definition of open-source.

Source: Own representation.

Criteria #1, #2, and #3 are adopted from A. Rahman (2018a). Nevertheless, #2 is slightly adapted to be the average commit number per month because the pre-analysis of the search results showed that some suitable repositories are not updated every month but are still well maintained. Criteria #5 and #6 are adapted from A. Rahman, Parnin, and Williams (2019). However, six contributors are set as the threshold instead of nine. This change is due to the pre-analysis that revealed suitable repositories with six contributors. Regarding criterion #7, only repositories that have a license specified are considered. Otherwise, they must not be used by others than the copyright holder (Häußge, n.d.).

3.4.2. Proprietary Repositories

In addition to the open-source repositories, proprietary infrastructure repositories are selected to increase the general validity of the findings. Since the author of this thesis works for Capgemini (CG) (https://www.capgemini.com), a ~ 350,000-employee company, many proprietary repositories of Capgemini are accessible. Some of the inclusion and exclusion criteria from Table 10 are also applied to the proprietary repositories of Capgemini. However, not all can be applied. Table 11 shows the inclusion and exclusion criteria for the proprietary repositories from Capgemini.

Table 11. Inclusion and Exclusion Criteria for the Selection of Proprietary Repositories From Capgemini

#	**Criterion name**	**Inclusion criterion**	**Exclusion criterion**
1	Commits per Month	≥ 2	< 2
2	IaC Proportion	≥ 11%	< 11%

Source: Own representation.

Since the source code must not be published, the infrastructure files can only be analyzed by the author of this work and the employees of Capgemini participating. As indicated above, the following research question guides this chapter: **RQ 3.4.2**: *For each selected IaC tool, what are two suitable proprietary repositories to conduct static code analysis?*

3.5. Assessment of Static Infrastructure Code Analyzers

So far, this work has created a list of IaC tools (chapter 3.2), SICAs (chapter 3.3), and suitable open-source and proprietary repositories to test the SICAs (chapter 3.5). Thereupon, the next step is to perform a qualitative assessment of the SICAs because "any static analysis tool is subject to evaluation" (A. Rahman, 2018a) to reveal potential strengths and weaknesses. Notably, the assessment of the SICAs does not aim to find *the* best tool because a tool can only be more or less suitable for a specific use case (as in Novak et al. (2010) or Ibnelbachyr (2021)). First, the number of IaC tools a SICA supports could be considered a quality metric. However, if a practitioner is only interested in analyzing their *Ansible* code, *checkov*, one of the most popular SICAs that supports many IaC tools, is unsuitable since it does not support *Ansible*. Second, the analyzed SICAs belong to different categories with different use cases. Hence, comparing tools of different categories does not make sense. The assessment of the SICAs aims to identify all characteristics of the SICAs under test to give researchers and practitioners support for future decisions. The assessment of the SICAs is based on quantitative metrics (e.g., number of GitHub stars) and qualitative analysis (e.g., quality of documentation) to give a meaningful overview of each SICA. The following four research questions guide the SICA assessment. The methodologies to answer the RQs are described in more detail in the following chapters.

- **RQ 3.5.1**: *What are the characteristics of the identified SICAs?*
- **RQ 3.5.2**: *How do practitioners assess the usefulness of the findings of the identified SICAs?*
- **RQ 3.5.3**: *How are the identified SICAs implemented?*
- **RQ 3.5.4***: Which IaC tools and use cases lack SICAs?*

3.5.1. RQ 3.5.1: What are the Characteristics of the Identified SICAs?

RQ 3.5.1 aims to establish a list of all characteristics of the SICAs that interest researchers and practitioners. To begin with, which characteristics describe a SICA? As mentioned in chapter 2.5, an essential characteristic of each static code analyzer is how easily it can be **integrated into the development process**, that is, into the local development environment (e.g., IDE) and the CI/CD pipeline. The earlier misconfigurations are identified, the easier they can be fixed. For instance, *checkov* offers a VS Code extension. According to the authors of *checkov* (bridgecrew, n.d.–a), this IDE extension aligns with the DevSecOps principle of shifting security left (PR Newswire, 2021).

A standard measure to quantify the popularity of a GitHub repository is its **number of GitHub stars**. Brikman (2022) also used the number of GitHub stars to assess SICAs. He furthermore used the following categories, which are all adopted in this work: **license**, **backing company**, **contributors**, **first release**, **latest release**, **built-in checks**, and **custom checks**.

Practitioners have compared SICAs in several articles and defined evaluation criteria that are important to them. These evaluation criteria are adopted and used in this work. Ibnelbachyr (2021) compared eight static infrastructure code analyzers for *Terraform*. The author analyzed the **programming language** of each SICA. Beyond that, he listed the **number of GitHub stars** and the **backing company**, if applicable. Beyond that, he investigated the **last release version.** Nevertheless, the latest release version does not imply how up-to-date a SICA is. This is because how the SICAs are versioned is up to the maintainers. Hence, this characteristic is not applied to the assessment in this work. Instead, the **last release date** is more interesting because it shows how up-to-date a SICA is. In addition to those criteria, Ibnelbachyr (2021) analyzed the SICAs' **built-in checks** and the **language, complexity, and expressiveness of custom checks**. Lastly, the author investigated the usability of the SICAs. For example, can checks be **ignored via source code or via the CLI**? How readable is the **output**? Is there a **VS Code extension**? Tafani-Dereeper (2020) took a similar approach by comparing different static infrastructure code analyzers for *Terraform* using slightly different SICAs. Compared with Ibnelbachyr (2021), the author also mentioned the number of maintainers in case a public GitHub repository is available. Nonetheless, the number of maintainers is difficult to quantify (it cannot easily be extracted from the repository). Since backers and the number of contributors are already included in the analysis, the number of maintainers is omitted. In addition to the already explained criteria, Tafani-Dereeper (2020) also measured the **speed** of the SICAs. Notably, a fair comparison between SICAs is only possible when they scan the same number of lines of code. Nath (2021) compared six SICAs in the GitHub repository iacsecurity (2021). In addition to the previously mentioned criteria, he analyzed the **licenses** under which the SICAs are available. Moreover, the author explicitly investigated the specific **CI/CD integrations**, like GitHub Actions. In addition, he offered test cases with which a **catch rate** per SICA can be quantified. Fábry (2021) compared four SICAs. In addition to all already mentioned criteria, the author analyzed which **IaC tools** each SICA supports, whether **specific issues can be ignored** (e.g., with comments), and whether checks can be specifically **blacklisted** or **whitelisted**. Moreover, Fábry (2021) investigated if the SICAs can **auto-fix** (i.e., automatically resolve) issues. His last criterion was about the existence of **documentation**. Since all four tools evaluated by Fábry (2021) are documented, the **quality of documentation** seems more meaningful. Table 12 gives an overview of the discussed criteria.

Table 12. Criteria for the Assessment of the Identified SICAs

Characteristic	**Description**
IaC Tool Support	Which IaC tools does the SICA support?
Development Support	How easily can the SICA be integrated into IDEs, version control (VC), and the CI/CD pipeline? Are there CI/CD-specific integrations? Is there a VS Code extension?
GitHub Stars	How popular is the SICA in the community?
License	What license applies to the tool? Is it free to use also in a commercial context?
Backing Company	Does a company back the tool? A backing company may indicate that the tool is well maintained but may also increase the risk that the tool will become commercial in the future.
Contributors	How many contributors does the repository have?
First Release	How old is the tool?
Last Release	When was the latest version of the tool published?
Built-in Checks	Does the tool have built-in checks?
Custom Checks	Does the SICA allow the definition of custom checks? What language must be used to create custom checks?
Ignoring Checks (Blacklisting)	Can checks be ignored via source code or the CLI?
Ignoring Specific Issues	Can specific issues be ignored, for instance, via comments?
Activating Checks (Whitelisting)	Can checks be whitelisted?
Output	Which output formats are supported (e.g., JSON)?
Speed	How fast is the tool?
Auto-Fix	Can the SICA automatically resolve issues?
Quality of Documentation	What is the quality of the documentation?

Sources: Own representation based on Brikman, 2022; Ibnelbachyr, 2021; Tafani-Dereeper, 2020; Nath, 2021; Fábry, 2021.

3.5.2. RQ 3.5.2: How do Practitioners Assess the Usefulness of the Findings of the Identified SICAs?

The criteria evaluated to answer **RQ 3.5.1** deliver important information about the SICAs. These help practitioners select suitable tools, and researchers and tool vendors find directions for future research. In addition to those criteria, the tools must be executed. Thereupon, the quality of the findings must be evaluated.

With that in mind, the SICAs selected for the in-depth assessment are executed against the code of the identified repositories (see chapter 3.4). Then, the findings are evaluated by two practitioners, and Cohen’s Kappa score (Cohen, 1960) is calculated, which measures the agreement between two raters. A. Rahman (2018a) deployed developers and maintainers of the analyzed repositories to assess the findings. One disadvantage of this approach is that developers tend to be biased about the quality of their code (Acar et al., 2017, pp. 164–165). Hence, in the present work, the SICAs’ findings in the open-source repositories are evaluated by independent developers. Still, developers of the respective CG projects evaluate the findings in the proprietary repositories because the code must not be shown to others. Calculating the agreement measure (Cohen’s Kappa score) is essential because the interpretation of misconfiguration is subjective (Hall et al., 2014, p. 7; Mäntylä & Lassenius, 2007, p. 395). The same procedure was applied by A. Rahman (2018a) to assess the *SLAC* even though the author had significantly more raters (89).

During the qualitative analysis, each tool is executed, and the findings are classified. In doing so, one of two classes is applied: **true positive** (a SICA identified an issue which, according to the two practitioners, actually is one) or **false positive** (a SICA identified an issue which, according to the two practitioners, is no issue). Information about SICAs’ false positives is vital because this is a known challenge for static analyzers (Bhuiyan & Rahman, 2020; Zampetti et al., 2017). Soundness is another term to describe how many false positives a tool produces (Schwarz et al., 2018). It gives an idea about the initial setup effort of the SICA. Beyond that, it is also essential to find out how many **findings practitioners would genuinely resolve**. This is because there is a difference between agreeing to a particular finding and investing time to resolve it.

The **true positive** and **false positive rates** reveal the quality of the issues the SICAs actually do find. Besides, what about those issues the SICAs do not find? These are called false negatives. The completeness of a SICA quantifies the occurrence of false negatives (Schwarz et al., 2018). It must be known beforehand which issues exist in a set of infrastructure manifests to measure the **false negative rate** and the **catch rate** of a SICA. Hence, this analysis is more complex and time-consuming and requires a benchmark repository where the existing issues are known. One such repository is *terragoat* (bridgecrew, 2020c) for *Terraform*, a so-called vulnerable-by-design repository. Fábry (2021) also uses *terragoat* in his work. Bridgecrew offers similar vulnerable-by-design repositories for *CloudFormation* and *kustomize*. However, this work analyzes neither the **catch rate** nor the **false negative rate** because it is out of scope.

Each practitioner is given a list with the findings of the SICAs and the code of the respective repository. The practitioners must then decide for each finding whether they agree, disagree, or do not know. If they disagree with a finding, they must explain why. Since the number of findings for a SICA and repository could be exceptionally high, only the first fifty findings are evaluated. Thereby, practitioners evaluate critical issues first while ignoring less severe findings.

For SICAs that support only one IaC tool, it is evident against which repositories they are tested. However, SICAs supporting multiple IaC tools (e.g., *checkov*) could be used on multiple repositories. Since this procedure would drastically complicate the in-depth assessment, the SICAs supporting multiple IaC tools are tested against repositories of the IaC tool for which they have the most rules. For instance, *checkov* would be executed on *Terraform* repositories because it has the most rules for *Terraform*. The SICAs are run in their default configuration, for instance, without disabling specific checks.

The described characteristics are summarized in Table 13, which, together with Table 12, completes the qualitative and quantitative assessment of the analyzed SICAs. As mentioned in chapter 3.3, if more than ten SICAs meet the inclusion criteria, scoring is performed to select SICAs for the in-depth assessment. All tools are evaluated with the criteria in Table 12. Beyond that, only the selected SICAs are evaluated with the criteria in Table 13.

Table 13. Criteria for the In-Depth Assessment of the Selected SICAs

Characteristic	**Description**
Fix Rate	How many findings would practitioners resolve compared to the total number of findings?
False Positive Rate	What is the false positive rate of the SICA?

Source: Own representation.

3.5.3. RQ 3.5.3: How are the Identified SICAs Implemented?

RQ 3.5.1 and **RQ 3.5.2** mainly help practitioners to decide on a particular SICA for their use case. Beyond that, researchers and tool vendors are interested in the implementation approaches of existing SICAs. Researchers and tool vendors can use these data if they decide to create new SICAs. **RQ 3.5.3** determines the implementation approaches of the analyzed SICAs.

First, the SICAs' **programming languages** suggest suitable choices for new SICAs. Researchers might choose a dominant programming language to take advantage of the benefits others saw. They might also choose a language not used so far for evaluation purposes. Only those programming languages that account for at least 10% of the code in a repository are considered. All others are most likely just configuration files or similar. Furthermore, it is not only of interest which rules the identified SICAs implement but also how these **rules are implemented**. Table 14 summarizes the analyzed implementation details.

Table 14. Criteria for the Analysis of the Implementation Details of the Identified SICAs

Characteristic	**Description**
Programming Languages	Which programming languages have been used to create the SICA?
Rule Implementation	How are the rules implemented?

Source: Own representation.

The collected data is made public online via the *IaC Analyzer Decision Guide*. As will be explained later (chapter 4.6), the data is available in the GitHub repository of this thesis. Making the dataset public is inspired by the research of Schermann et al. (2018).

3.5.4. RQ 3.5.4: Which IaC Tools and Use-Cases Lack SICAs?

In essence, **RQ 3.5.1**, **RQ 3.5.2**, and **RQ 3.5.3** describe the characteristics of the identified SICAs. This data can be used to identify areas not already covered by existing SICAs, that is, understudied areas. Understudied areas could be IaC tools and categories that cannot be scanned with any identified SICA. It could also be an area where a SICA exists, but the SICA shows significant disadvantages. For instance, a SICA could be archived. Based on this analysis, researchers can guide their future research. The information about the understudied areas of static infrastructure code analysis is derived from the findings of chapters 3.5.1, 3.5.2, and 3.5.3. Based on these data, a matrix illustrating the understudied areas is created, which is one main deliverable of this work.

After all, the data collected in chapter 3.5 must be provided to practitioners and researchers in a suitable format. For this purpose, the following chapter elaborates on the methodology to develop such a platform.

3.6. Design and Implementation of the Static IaC Analyzer Decision Guide

The outcome of chapter 3.5 must be transferred into a more suitable communication format for practitioners. Publishing an Excel document listing all SICAs and their characteristics would hardly benefit practitioners. The Excel document would be time-consuming to study. Thus, this chapter describes the design and development of the *IaC Analyzer Decision Guide* with the design science research (DSR) methodology. Andrzejewski et al. (2017) also use DSR to implement a decision guide. Although the authors do not work on a pure IT-related topic, it shows that design science research is an appropriate methodology for this thesis.

Österle et al. (2011, p. 9) define four fundamental principles each DSR project must follow. These principles distinguish a DSR project clearly from the practitioners' community and commercial providers. First, an artifact must apply to a class of problems (abstraction). Second, the research project must provide a valuable contribution to the existing scientific knowledge base (originality). Third, an artifact created in the DSR process must be explainable and evaluable (justification). Fourth, such an artifact must benefit the defined target group (benefit). All of these fundamental principles are met for the design and development of the *IaC Analyzer Decision Guide*. The design of the *IaC Analyzer Decision Guide* applies to other decision guides and can easily be transferred to similar use cases (abstraction). Also, since there is no such tool at the time of writing, it contributes to the scientific knowledge base by encoding and combining knowledge about IaC tools and corresponding static analyzers (originality). As with every software application, the developed tool is evaluable, for instance, via usability test sessions (justification). Last, the application benefits both practitioners and researchers because it makes the knowledge more accessible and, due to its design, allows it to be easily extended and updated (benefit).

Many variations of the DSR methodology exist (Benner-Wickner et al., 2020). Equally, all existing DSR methods consist of three cycles: relevance cycle, design cycle, and rigor cycle (Benner-Wickner et al., 2020). Each cycle acts as an input for the following. The entire regulative cycle (relevance cycle, design cycle, rigor cycle) is performed iteratively.

- **Relevance Cycle:** The relevance cycle is based on the application domain's people, organizations, and systems. The problems in a particular application domain lead to the definition of research in this area. Also, the problem at hand is used to define the criteria for evaluating the developed solution. (Benner-Wickner et al., 2020)
- **Design Cycle:** The design cycle is driven by the development of the artifact (Benner-Wickner et al., 2020).
- **Rigor Cycle:** The rigor cycle extracts relevant information from prior research. This data is incorporated into the design cycle. Furthermore, a significant responsibility of the rigor cycle is communicating the findings (Benner-Wickner et al., 2020).

The regulative cycle consisting of relevance, design, and rigor cycle is relatively abstract. In contrast, Benner-Wickner et al. (2020) explained that the DSR definition of Österle (2010) is a pragmatic version of it. Österle (2010) emphasized that their approach is more design-oriented than other behavioristic-oriented approaches. **Thus, the design science research implementation in the present work is oriented towards Österle (2010).** Each of the four process steps explained by Österle (2010) is described in more detail in the following sub-chapters. The four steps of analysis, design, evaluation, and diffusion are performed iteratively.

Following the guidelines of Hoang Thuan et al. (2019), this work aims to develop a concrete instantiation of an information system. With the help of the research questions templates of Hoang Thuan et al. (2019), the overall research question of this entire chapter is defined as follows. **RQ 3.6**: *How can the collected data about static infrastructure code analyzers be used to develop an application which supports practitioners in finding suitable tools for their specific requirements?*

3.6.1. Analysis

The analysis phase defines the **practical problem**, the **research objectives**, the **research questions**, the **research gap**, and a **plan to develop the artifact** to solve the problem (Österle, 2010, p. 4). Furthermore, the **requirements**, **target group**, and **roles** must be defined (Benner-Wickner et al., 2020, p. 7). The analysis phase also investigates the state-of-the-art of related problem-solving approaches (Österle et al., 2011, p. 9). In addition to Österle's work (2010), Knauss (2021) explicitly created a set of guidelines for DSR applied in Master’s theses. Hence, these guidelines are applied in addition to the procedure defined by Österle (2010). Following the advice of Knauss (2021), the artifact must be clearly **defined right at the beginning** to set the scope (Knauss, 2021, p. 114), which is done in the analysis phase. Moreover, the author states that DSR projects should comprise a maximum of **three iterations** in which the artifact is constantly improved. Hence, this work does not exceed three iterations. Nevertheless, the flaws identified in the third evaluation cycle are still corrected. For each phase of the regulative cycle (relevance cycle, design cycle, rigor cycle), a research question must be established (Knauss, 2021, p. 115). Since Österle's (2010) DSR approach is applied in this work, one research question is established for each of the four phases (analysis, design, evaluation, diffusion), not each phase of the regulative cycle. On the whole, the research question for the analysis phase is (oriented toward Knauss (2021, p. 115)): **RQ 3.6.1**: *What is the current problem with finding suitable static code analyzers for an infrastructure project?*

3.6.2. Design

The requirements defined in the analysis phase and the literature review results are incorporated into the artifact’s design (Benner-Wickner et al., 2020, p. 7). The design is accomplished using accepted methods in the scientific community (Österle, 2010, p. 4). Besides, the artifact is permanently checked against the established requirements to guarantee that it fulfills its purpose (Benner-Wickner et al., 2020, p. 7). In short, the research question for the design phase is (oriented toward Knauss (2021, p. 115)): **RQ 3.6.2**: *How can the problem of finding suitable static code analyzers be overcome?*

3.6.3. Evaluation

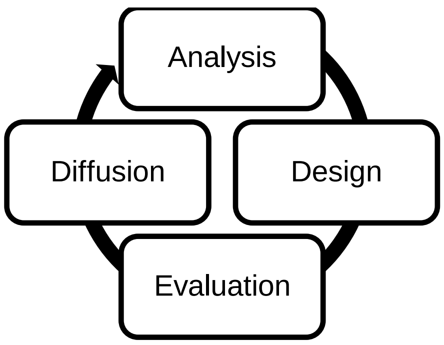
The created artifact is evaluated against the research goal defined in the analysis phase (Österle, 2010, 4p). It must be tested whether the developed artifact solves the practical problem. Also, the artifact is, once more, systematically verified against the requirements from the analysis phase (Benner-Wickner et al., 2020, p. 7). A widespread method to test such a software application is by conducting usability test sessions with users of the target group. This validation approach is also chosen for this thesis, that is, by conducting **usability tests** with practitioners who work on IaC projects. The participants’ consent is obtained through the consent form in Appendix A. The informed consent declaration (Appendix A), usability test plan (Appendix B), script (Appendix C), and report (Appendix D) are adapted from Niels (2021). Usability tests “determine the extent to which the software product is understood, easy to learn, easy to operate, and attractive to the users” (Black, 2015, chapter 4). The evaluation result is then used to improve the application's usability. To sum up, the research question for the evaluation phase is (oriented toward Knauss (2021, p. 115)): **RQ 3.6.3**: *How does the artifact support practitioners and researchers in finding existing SICAs?*

3.6.4. Diffusion

After evaluating the artifact, the learnings are generalized to other domains. Also, the result of the research project shall be made accessible to the target groups, for instance, via a website (Österle, 2010, p. 16) or open-source software (Benner-Wickner et al., 2020, p. 8; Österle et al., 2011, p. 9). The outcome of this work must be accessible to both target groups – researchers and practitioners. To summarize, the research question for the diffusion phase is: **RQ 3.6.4**: *What learnings can be applied to similar problems?*

Figure 6 shows the four phases of design science research according to Österle (2010).

Figure 6. The Four Phases of Design Science Research



Source: Own representation based on Österle, 2010.

3.7. Summary of the Applied Methodology

This chapter summarizes the entire research process shortly. First, public cloud providers are identified. These are referred to in the next chapter, where provisioning tools for the selected cloud providers and IaC tools for all other categories are investigated. The catalog of IaC tools is then used in the next chapter to find static code analysis tools for those IaC tools. Based on the selected IaC tools, in the next chapter, open-source and proprietary repositories are identified. Then, the identified SICAs are assessed. For instance, it is analyzed which IaC tools they support. Also, the SICAs selected for the in-depth assessment are executed against the previously identified repositories. The outcome of this chapter, data about the SICAs and their corresponding characteristics, is then used to establish a tool suitable to communicate the findings to practitioners and researchers.

4. Research Findings

This chapter elaborates on the research findings achieved with the previously outlined methodologies. At the end of each chapter, a summary answer to the respective research question defined in chapter 3 is given. The detailed research material is available at https://github.com/nileger/iac-analyzers/tree/main/thesis. To begin with, the most popular public cloud providers are identified.

4.1. Identification of Public Cloud Providers

The five most popular cloud providers are **Amazon Web Services, Microsoft Azure, Google Cloud, Oracle Infrastructure Cloud, and IBM Public Cloud** (Scribd, 2022, March 8). Thus, chapter 4.2 identifies provider-specific provisioning tools for those five cloud providers. Provisioning tools for other cloud providers are omitted. As explained earlier, all other IaC tool categories are cloud provider-independent. According to Armstrong (n.d.), smaller cloud providers prefer building *Terraform* providers instead of creating IaC tools like *AWS CloudFormation*. Nevertheless, since only popular public cloud providers are considered, it is likely that they also have their own IaC tools.

The identified public cloud providers are now shortly explained and analyzed regarding their usage options. All identified public cloud providers have a CLI, which can be used to create, update, and delete resources in the cloud. Those CLI commands can be executed ad-hoc by developers or embedded in scripts. All cloud providers except Google Cloud and Microsoft Azure have cloud development kits (CDKs). Developers use CDKs to manage cloud resources with general-purpose programming languages like Java, JavaScript, or Go. These languages are often more familiar to the involved developers. Also, it allows the development of applications on top of the cloud providers' capabilities.

A summary of the public cloud providers and their usage options is provided in Table 15. The answer to the **research question** about the five most popular cloud providers (*What are the top five most popular public cloud providers?*) is AWS, Azure, Google Cloud, Oracle Infrastructure Cloud, and IBM Public Cloud.

Table 15. The Five Most Popular Public Cloud Providers

Name	**CLI**	**CDK**
AWS	Yes	Yes
Azure	Yes	No
Google Cloud	Yes	No
Oracle Infrastructure Cloud	Yes	Yes
IBM Public Cloud	Yes	Yes

Sources: Own representation based on Scribd, 2022, March 8.

4.2. Identification of IaC Tools

In the previous chapter, the most popular public cloud providers have been identified. Thereupon, existing IaC tools are identified in this chapter. Whereas provisioning tools are only considered for the public cloud providers identified in chapter 4.1, all other tools are cloud provider-independent. Additionally, categorization approaches for IaC tools are explained.

4.2.1. RQ 3.2.1: How do Practitioners and Researchers Categorize IaC Tools?

During the analysis of the selected resources, IaC tool categories used by practitioners and researchers were extracted to answer **RQ 3.2.1** (*How do practitioners and researchers categorize IaC tools?*). Forty informal sources mentioned at least two of such categories. Most of the sources differentiated between declarative and imperative IaC tools. Also, the classification into mutable and immutable, procedural (imperative) and declarative, master and masterless, agent and agentless, large community and small community, and mature and cutting-edge, as outlined by Brikman (2022), were used among practitioners. In addition to those categories, practitioners differentiated between tools for the initial setup phase and the maintenance phase of infrastructure. Since all these mentioned categories were known before the MLR, they do not suggest new search strings for the MLR. Equally, the ten formal resources that outlined IaC tool categories cannot be used to create new search strings.

4.2.2. RQ 3.2.2: What IaC Tools Exist?

Summary of the Results

In total, 238 sources were analyzed. On average, each source mentioned 6.4 alleged IaC tools. Each IaC tool mentioned is referred to as one vote in the following. Beyond the 238 analyzed sources, 80 sources were skipped because they did not mention any IaC tool, were duplicates, or belonged to tool vendors who only advertised their own products. One hundred ninety-six different IaC tools were referenced, on which the 1,538 votes were distributed. Of the 196 IaC tools, 45 (23%) meet the requirements of this work. *Ansible* (161), *Puppet* (146), and *Terraform* (145) are the three tools with the most votes, whereas multiple tools have less than five votes (24 tools). Some of the mentioned tools that do not meet the inclusion criteria are not specific to IaC but can be used to manage IaC-related topics. For instance, practitioners and researchers use GitHub, GitLab, and Bitbucket to collaborate on IaC projects. They also apply Jenkins, GitHub Actions, or Circle CI to provision infrastructure via IaC tools. However, none of these tools (77%) are IaC tools.

Ten of the 45 considered tools are configuration management tools: *Chef*, *Puppet*, *Ansible*, *SaltStack*, *CFEngine*, *R?ex*, *cloud-init*, *cdist*, *mgmt*, and *SmartFrog*. Two times fewer (five) server templating tools were identified in the MLR: *Docker*, *Packer*, *Vagrant*, *NixOs*, and *Azure VM Image Builder*. In contrast to the rare server templating tools, plenty of orchestration (12) and provisioning tools (17) were identified. The discovered orchestration tools are *Kubernetes*, *Swarm*, *Nomad*, *Docker Compose*, *Podman*, *AWS Copilot*, *Apache Brooklyn*, *Fargate*, *Hyscale*, *AWS Serverless*

Application Model (SAM), *Azure Functions*, and *Bolt*. The identified provisioning tools are *Terraform*, *CloudFormation*, *Azure Resource Manager*, *Bicep*, *Google Deployment Manager*, *OpenStack Heat*, *Terragrunt*, *Pulumi*, *AWS CDK*, *Serverless Framework*, *Opta*, *CDKTF*, *Cloudify*, *Bosh*, *Charon*, *Argon*, and *Architect*. The only package manager mentioned is *Helm*. All the identified tools are listed below, which is also the answer to this chapter's research question (**RQ 3.2.2**): *What IaC tools exist?*

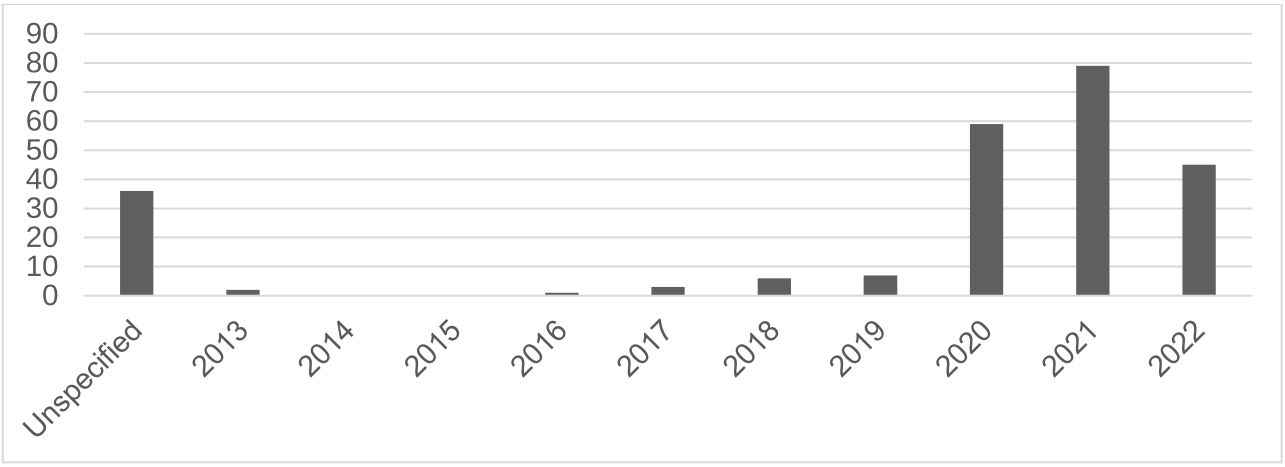
- **Configuration Management:** *Chef, Puppet, Ansible, SaltStack, CFEngine, R?ex, cloud-init, cdist, mgmt, SmartFrog*
- **Server Templating:** *Docker, Packer, Vagrant, NixOs, Azure VM Image Builder*
- **Orchestration:** *Kubernetes, Swarm, Nomad, Docker Compose, Podman, AWS Copilot, Apache Brooklyn, Fargate, Hyscale, AWS Serverless Application Model (SAM), Azure Functions, Bolt*
- **Package Managers:** *Helm*
- **Provisioning:** *Terraform, CloudFormation, Azure Resource Manager, Bicep, Google Deployment Manager, OpenStack Heat, Terragrunt, Pulumi, AWS CDK, Serverless Framework, Opta, CDKTF, Cloudify, Bosh, Charon, Argon, Architect*

Arguably, *Kubernetes*, *Docker Compose*, and *Swarm* may be considered container orchestration platforms. However, these tools have their own manifests to describe the deployed resources. Since those manifests are written as code and can be subject to code smells and security issues, they are considered IaC tools too. Furthermore, the number of votes collected in the MLR clearly shows that practitioners and researchers consider them IaC tools. Whenever those tools are mentioned in this work, the manifests are meant, not the platform itself.

Analysis of the Sources

As described in chapter 3.2, only sources from the past two years were selected. Hence, most sources (76%) were created between 2020 and 2022, considering sources with a specified date. Thirty-five sources did not have a date specified in the article, even though the Google search engine judged those articles to have been published between 2020 and 2022. Trusting the Google search engine, approximately 91% of the analyzed sources fall into the defined time range. The other 9% of sources were distributed between 2013 and 2019 and were primarily identified via snowballing in formal literature or found during the initial literature review. For instance, an article about *Charon* was published in 2013 (Dolstra et al., 2013). Figure 7 illustrates the described distribution of sources per year. It shows that the analyzed sources are up-to-date. Hence, the results described above reflect the state of research and -practice.

Figure 7. IaC Tools: Number of Sources per Year

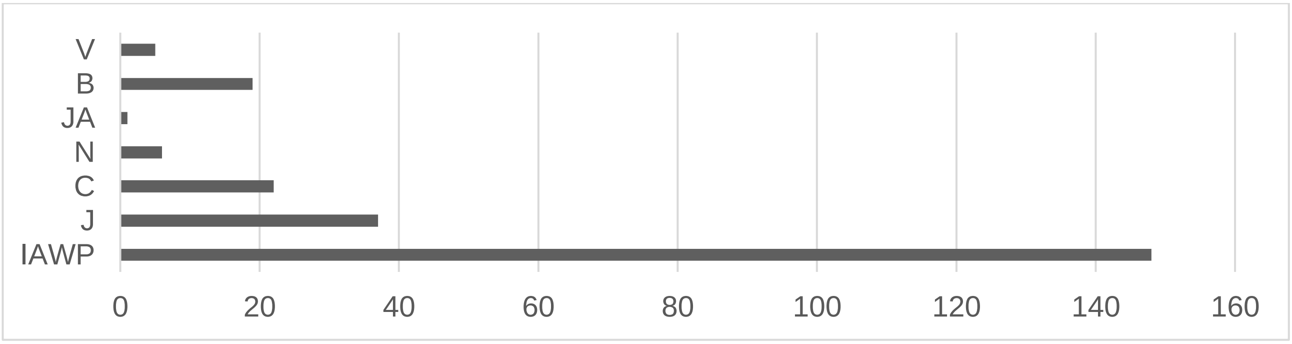


Source: Own representation.

In addition to the distribution of the selected sources per year, it is also interesting to investigate the source types. This is because the multivocal literature review considers all source types, not only formal literature. Figure 8 shows that the majority (62%) of sources are internet articles or white papers (IAWP), followed by journal articles (J) (16%) and conference papers (C) (9%). Moreover, books (B), newspaper articles (N), job advertisements (JA), and videos (V) were analyzed. In summary, about ¾ of the sources are informal, and ¼ are formal literature. As stated in Garousi et al. (2019), the ratio of informal literature (62% (IAWP) + 8% (B) + 3 % (N) + 1% (JA) + 2% (V) = 76%) in this chapter falls well between the reported boundaries of other MLRs ranging from 14.3% to 85.9%, especially when considering that IaC is a highly practitioner-oriented topic.

The contribution of the formal literature analyzed is twofold. First, many research studies created new IaC tools, like *Charon* (Dolstra et al., 2013). Second, the research studies helped to develop a clear picture of the most popular IaC tools. For example, *Ansible*, *Puppet*, *Chef*, and *Terraform* were the IaC tools mentioned by the most formal research studies. Likewise, this observation shows that IaC research mainly focuses on configuration management tools and only rarely considers other tools like *Docker* (e.g., Schermann et al. (2018)). Even though the large quantity of IAWP resources mentioned many tools that are not IaC tools (e.g., Jenkins), these articles also helped to identify several IaC tools that are unpopular in the formal literature, like *mgmt* (Shubin, 2015).

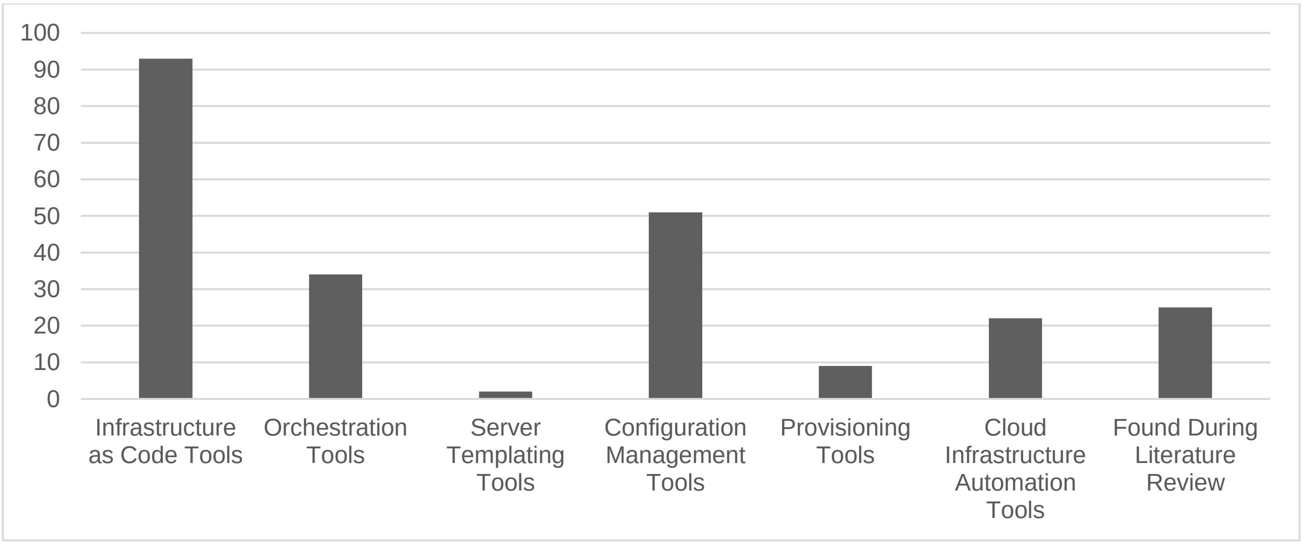
Figure 8. IaC Tools: Number of Sources by Source Type



Source: Own representation.

Various search strings were applied to search for existing IaC tools, and the number of sources found varies significantly between these search strings. Figure 9 shows that searching with *Infrastructure as Code Tools* led to the most results (39%), whereas the search string *Server Templating Tools* contributed only 1% of the selected sources. *Configuration Management Tools* (22%) and *Orchestration Tools* (14%) gave the second and third most results. Interestingly, the search string *Provisioning Tools* contributed only 4% even though *Terraform* (145 votes), *AWS CloudFormation* (97 votes), and several other highly voted tools belong to this category. In total, 35% of the votes are for provisioning tools (Figure 10). These findings suggest that provisioning tools are rarely referenced by their IaC tool category name. Likewise, the findings indicate that the term S*erver Templating Tools* is not widely adopted. *Configuration Management Tools* gives the most results among all categories, which is well in line with the literature analyzed in chapter 2: Most research studies mention configuration management tools when talking about infrastructure as code. Beyond the literature discovered with these search strings, 11% of the analyzed resources were found during the initial literature review in chapter 2.

Figure 9. IaC Tools: Number of Sources by Search String



Source: Own representation.

It is not only of interest to associate search strings with the number of results but also how these results are distributed between formal and informal literature. Table 16 lists all applied search strings and the corresponding number of results in formal and informal literature. In formal literature, searching for *orchestration tools* (141) yields the most results behind *configuration management tools* (51) and *provisioning tools* (30). In contrast, searching for *server templating tools* yields no results in formal literature, at least for the specific inclusion and exclusion criteria. The results in informal literature are very similar. Searching for *configuration management tools* and *orchestration tools* yields many results too. Notwithstanding, most results were identified by searching for *infrastructure as code tools* (100).

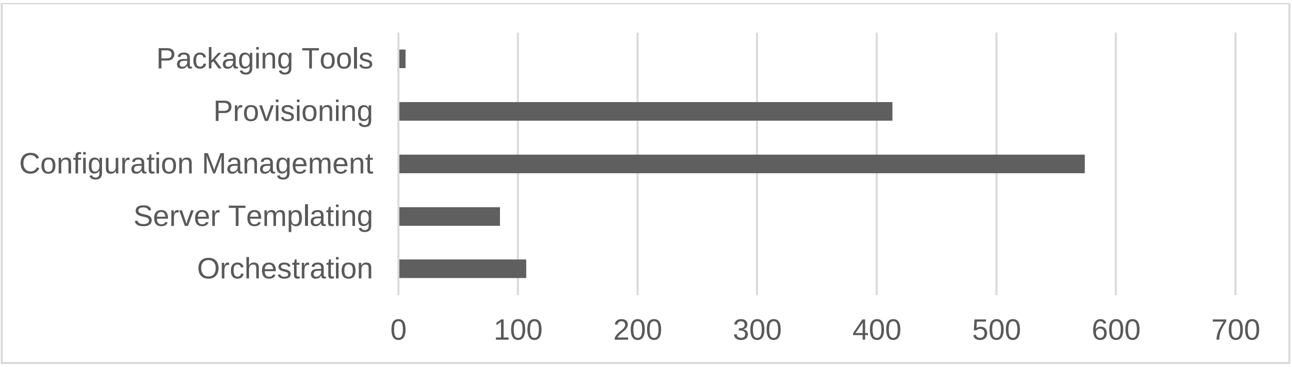
Table 16. IaC Tools: Search Strings and Stopping Criteria

Search String	**Formal: Number of Results**	**Informal: Stopped after N results**
Infrastructure as Code Tools	20	100
Orchestration Tools	141 (30 reviewed)	30
Server Templating Tools	0	10
Configuration Management Tools	51 (30 reviewed)	60
Provisioning Tools	30	10
Cloud Infrastructure Automation Tools	3[7]	40

Source: Own representation.

The distribution of votes per category is shown in Figure 10. It was already mentioned that 35% of the total votes were given for *provisioning tools*. Moreover, unsurprisingly, almost 50% of the votes were given to *configuration management tools*. This observation aligns with the literature review's impression that most formal literature covers configuration management tools. *Orchestration tools* and *server templating tools* contribute 9% and 7%, respectively. Votes for *packaging tools* (1%) are rarely given.

Figure 10. IaC Tools: Votes per Category



Source: Own representation.

Furthermore, of all 1,538 votes, 77% are considered good, meaning these votes were given for tools that meet the inclusion criteria of this work. Notably, 134 of the 151 IaC tools that do not meet the inclusion criteria have less than five votes. This observation suggests that those tools can be ignored confidently. The remaining 17 tools with more than five votes which do not meet the requirements were double-checked by the author and an IaC expert. Thereupon, it was confirmed that they did not meet the inclusion criteria. The high number of tools that do not meet the inclusion criteria for IaC tools (151 of 196) suggests asking why the analyzed sources were wrong in so many cases. The explanation lies in the applied methodology. During the multivocal literature review, the author

[7] For this particular search, the search target was set to keywords. Otherwise, more than 15,000 results were found.

extracted all tools he could find in the analyzed sources without pre-filtering. This was due to the enormous amount of sources. He could not read each article entirely, but he only quickly scanned the content for IaC tools. It could be that those tools now excluded were mentioned in another context. Therefore, articles contributing bad votes are not necessarily of bad quality. Also, some of the disqualified tools are IaC tools but do not meet the restrictions of this work. For instance, these tools are not open-source or have been last updated more than one year ago.

Still, many disqualified tools are not IaC tools. For instance, they are not specific to infrastructure (e.g., Shell scripts), have only a CLI and no code manifests, and are no longer maintained, endpoint tools software, or application configuration management tools. In particular, searching for *Configuration Management Tools* presented many articles with endpoint management tools or application configuration management tools. In essence, the high number of bad votes is due to the applied methodology, not necessarily the articles' quality.

Scoring

Although only 45 of the 196 identified tools meet the inclusion criteria, it is infeasible to use all for further analysis. Hence, scoring was applied to select each category's top four IaC tools. It was assumed that more popular tools are more likely to be covered by static analyzers developed by the community. This assumption implies that no statement about less popular IaC tools is possible. Still, this shortcoming was accepted in favor of better results for the most popular tools, which benefit most readers. For simplicity, the number of votes collected in the MLR was chosen as the scoring criterion, and the top four tools of each category were selected, given that a tool had more than five votes (Table 17).

Table 17. The Selected IaC Tools

Name	Category	Votes
Ansible	Configuration Management	158
Puppet	Configuration Management	144
Chef	Configuration Management	134
SaltStack	Configuration Management	85
Terraform	Provisioning	144
CloudFormation	Provisioning	96
ARM	Provisioning	46
Pulumi	Provisioning	39
Kubernetes	Orchestration	55
Docker Swarm	Orchestration	21
Nomad	Orchestration	8
Docker Compose	Orchestration	7
Docker	Server Templating	38

Name	Category	Votes
Vagrant	Server Templating	26
Packer	Server Templating	18
Helm	Packaging	6

Source: Own representation.

4.3. Identification of Static Infrastructure Code Analyzers

Forty-five IaC tools have been identified, of which 16 were selected for further analysis. With this in mind, this chapter discovers static analyzers for these tools.

4.3.1. RQ 3.3.1: How do Practitioners and Researchers Categorize SICAs?

During the multivocal literature review, **RQ 3.3.1** (*How do practitioners and researchers categorize SICAs?*) was answered by identifying the categories practitioners and researchers used to differentiate SICAs. Twelve practitioner sources mentioned such categorization approaches. Nevertheless, some of those classification approaches were only to differentiate the static analyzers from other testing approaches. Hence, only those sources describing different types of SICAs are explained in the following.

Salazar (2021) differentiated the SICAs into built-in commands, linters, SAST, and policy-as-code tools. Beyond that, Fábry (2021) used the categories formatting and linting, testing frameworks, and security scanning. The SICA *ValidIaC* categorizes its checks into validation (i.e., linting), security, cost, and map (visualizing the resources). Taptu (n.d.) took another categorization approach into multi-platform, *CloudFormation*, containers, *Terraform*, *Kubernetes*, *Ansible*, and *secret scanning*. Analysis Tools (2021) chose a similar method and used the following categories: *Ansible*, *configuration management*, *config files*, containers, *Kubernetes*, meta-linter, Security/SAST, template languages, and *Terraform*. Warkhade (2022) used the categories validation and security scanning to differentiate the SICAs, whereas Parsick (2020) differentiated linting and functional testing. In addition to the informal literature, seven formal sources mentioned static analysis tool categories. For instance, Chiari et al. (2022, p. 219) categorized SICAs into “linters, detectors of code smells and metrics that correlate with defects in IaC scripts.” For the most part, researchers distinguish SICAs into linting, formatting, compliance, built-in, and security tools.

The analyzed sources suggest various patterns to differentiate SICAs. First, SICAs are distinguished based on the **IaC tools** they support. Still, some tools cannot be assigned to a single IaC tool because they support multiple IaC tools. Second, practitioners categorized SICAs based on their functionality. They use the categories **formatting and linting (validation), security, cost prognosis, and resource visualization**. Third, SICAs were differentiated based on whether they are **built into** the IaC tool or are **third-party** tools. All these categorization approaches suggest characteristics that are decisive for a SICA. Hence, all characteristics are incorporated into chapter 4.6.

4.3.2. RQ 3.3.2: What SICAs Exist?

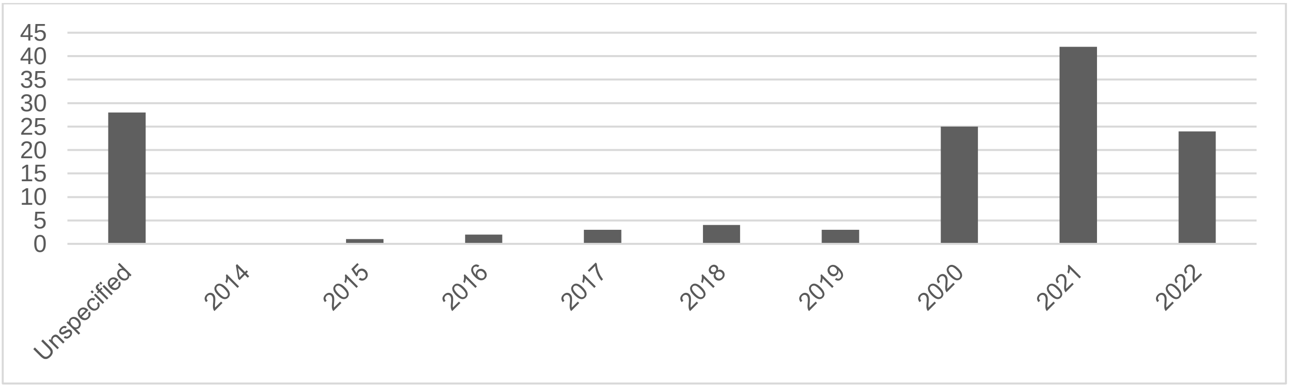
Summary of the Results

In total, 153[8] SICAs were identified via the multivocal literature review. 44 (29%) of those met the inclusion criteria and were subject to further analysis in chapter 4.5. Out of all tools which met the criteria, the tool with the most GitHub stars is *Trivy* (12,728), whereas *checkov* (41) has by far the most votes, followed by *tfsec* (31) and *terrascan* (21). The popularity of *Terraform* SICAs was foreseeable since *Terraform* is one of the IaC tools with the most votes, that is, one of the most popular IaC tools (see chapter 4.2). Hence, the ecosystem around it is more mature than it is for other IaC tools. It is also surprising that none of the formal research studies developed a tool popular among practitioners. For instance, *SLIC* and *SLAC* have few stars on GitHub compared to tools developed by practitioners. In summary, the following SICAs meet all inclusion criteria: **terraform validate, tflint, puppet parser validate, trivy, tfsec, cfn-lint, conftest, kube-score, infracost, terratest, OPA, molecule, ansible-lint, checkov, terraform-compliance, terrascan, cfn_nag, ansible-later, helm lint, puppet-lint, super-linter, mega-linter, cookstyle, hadolint, ansible --syntax-check, ValidIaC, ValidKube, Regula, terraform fmt, KubeLinter, KICS, semgrep, cloudformation-guard, prancer, cfripper, inframap, shisho, datree, kubeconform, polaris, kubescape, parliament, docker-compose-viz, and salt-lint.** This list answers this chapter's research question (**RQ 3.3.2**: *What SICAs exist?*).

Analysis of the Sources

Due to the defined inclusion criteria, most sources are from 2020 to 2022. However, older sources from formal literature were also considered because the body of static IaC analyzer literature is small. About 25 sources were analyzed that did not specify an exact date. Regardless, the Google search engine ranked those sources between 2020 and 2022. With this in mind, they were selected for the literature review. Figure 11 illustrates the number of sources per year and clearly shows that most articles are from 2020 or newer.

Figure 11. SICAs: Number of Sources per Year

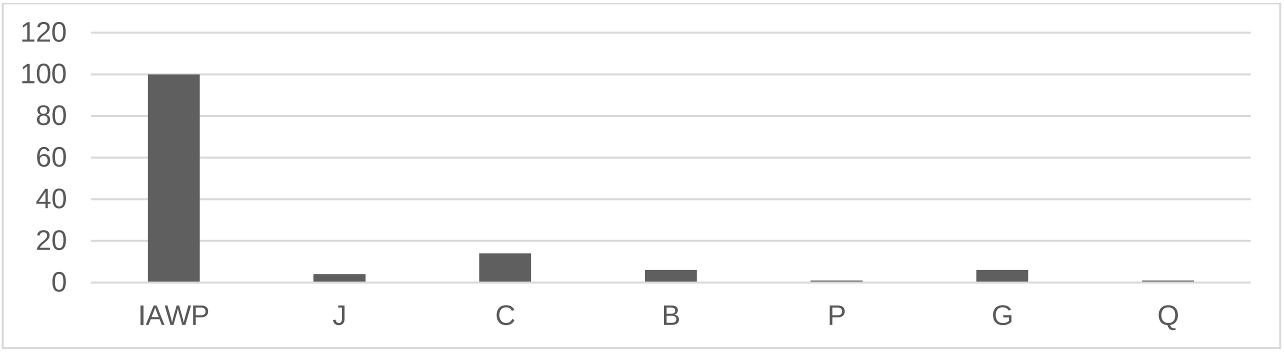


Source: Own representation.

[8] This includes the tools explored in chapter 4.5.4.

Similar to the multivocal literature review on IaC tools, the review on SICAs was dominated by internet articles or white papers (IAWP), contributing 76% to the analyzed resources. Compared to the MLR on IaC tools, this is a 14% increase and underlines the assumption that only little formal literature exists on SICAs, and that the topic is highly practitioner-oriented. Another explanation could be that critical search strings were missed in this work. Books (B) (5%), grey literature (G) (5%), conference papers (C) (11%), journal articles (J) (3%), Q&A sites (Q) (1%), and presentations (P) (1%) comprise the rest of the analyzed literature, which is illustrated in Figure 12.

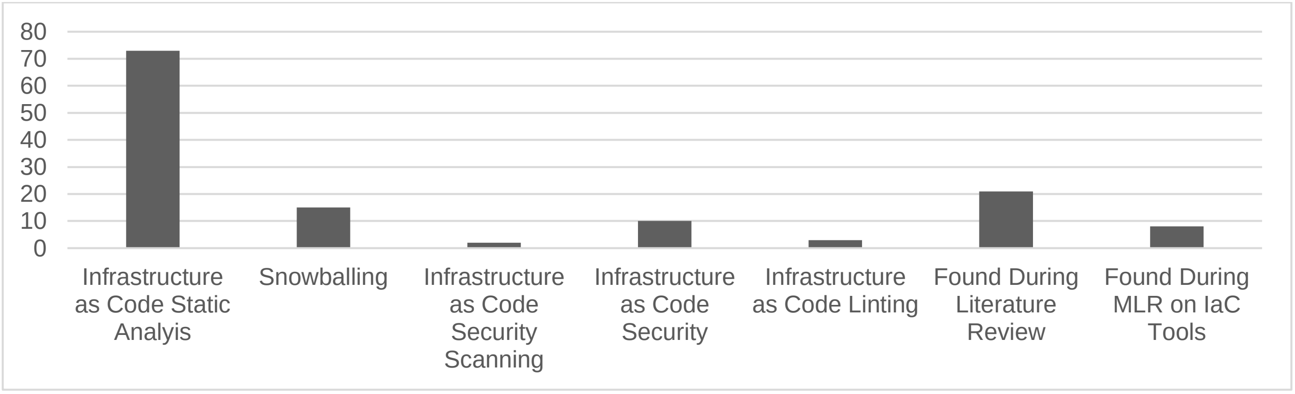
Figure 12. SICAs: Number of Sources by Source Type



Source: Own representation.

Most sources (73) were identified with the search string *Infrastructure as Code Static Analysis*, whereas all other search strings contributed significantly fewer sources. During identifying static infrastructure code analyzers, the author found it particularly helpful to check the README and documentation pages of the SICAs to scan for alternative SICAs or SICAs that those tools integrate. The author identified eight additional SICAs with this methodology that would have been missed otherwise. Those SICAs are *dockerlint*, *dockerfile_lint*, *dockerfilelint*, *kubeconform*, *rolespec*, *polaris*, *kubectl-neat*, and *kubescape*. Nevertheless, not all these tools met the inclusion criteria. This approach is comparable to snowballing in formal literature reviews. In total, fifteen sources were identified with snowballing, implying that no specific search string was used. Twenty-one sources were found during the literature review, and eight sources were already known from the MLR on IaC tools. An overview of the number of sources by search string is illustrated in Figure 13.

Figure 13. SICAs: Number of Sources by Search String



Source: Own representation.

Formal literature was searched via the search string *Infrastructure as Code* ˄ (*Static Analysis* ˅ *Security Scanning* ˅*Security* ˅*Linting)*. The formal literature search targeted keywords because too many articles would have been identified without this restriction. For instance, 21,376 articles would have been discovered for the first search string. Since Chiari et al. (2022) already conducted a formal literature review on static infrastructure code analysis, it was unlikely that additional SICAs would be identified during the formal literature review in this thesis. This assumption turned out to be true. Interestingly, none of the tools identified by Chiari et al. (2022) meets this work's requirements, especially considering the popularity of the tools and their state of development. These tools most often have less than ten GitHub stars. If they have more than ten stars, they have not been updated for more than one year. Still, their implemented rules could be transferred into other, more popular SICAs. This topic could be picked up by future work. To conclude, Table 18 overviews the applied search strings and the corresponding stopping criteria for formal and informal literature.

Table 18. SICAs: Search Strings and Stopping Criteria

Search String	**Formal Literature**	**Informal Literature**
Infrastructure as Code ˄ Static Analysis	9	100
Infrastructure as Code ˄ Security Scanning	0	10
Infrastructure as Code ˄ Security	33	20
Infrastructure as Code ˄ Linting	0	10

Source: Own representation.

Vote Quality

54% of all votes for SICAs were considered good votes. In contrast, in the MLR on IaC tools, 77% were considered good votes. Even so, this observation does not mean that the analyzed sources are of poorer quality. The reason for the lower vote quality is in the applied methodology. The author included all mentioned tools in the Excel sheet at the beginning of the MLR. This also applied to tools for which it was evident that they did not meet the requirements of this work. For instance, the author included *htmllint* (Coleman et al., 2014), an HTML linter. Up to a point, it became apparent that this approach would incorporate too many non-SICAs. After that, the author pre-filtered the mentioned tools. In essence, many articles mention IaC testing tools and container scanning tools, which are not static analyzers. 46% of the votes are considered bad votes – votes for tools that do not meet the requirements for SICAs in this work.

Tools Considered

Due to the limited time, not all SICAs could be tested comprehensively. Hence, it was decided to **exclude secret-scanning tools** to limit the number of SICAs. Although these tools are beneficial for infrastructure repositories because they are likely to contain sensitive information like AWS credentials, they are neither specific to infrastructure code repositories nor IaC tools. These tools work very similarly to linting tools for YAML or JSON. Due to this exclusion criterion, five tools (*GitLeaks*, *git-secrets, TruffleHof, detect-secrets, repo-supervisor*) were removed from the SICA list.

Ultimately, after applying the inclusion criteria to the list of SICAs, only 44 (29%) of the 153 collected tools were considered for further analysis. Most tools excluded were commercial, not specific to IaC, not maintained anymore, or did not perform static analysis. Also, as mentioned earlier, some of the tools developed by researchers, like *SLIC* or *SLAC*, did not meet the popularity requirements, especially the number of GitHub stars.

Scoring

All 44 SICAs were analyzed regarding their characteristics, but not all could be used for the in-depth assessment. This paragraph describes the procedure for selecting the SICAs for the in-depth assessment.

First, all SICAs for IaC tools not selected in chapter 4.2 were removed. Tools that do not specifically support any IaC tool but can be used to create custom rules (e.g., *OPA*) were still considered. However, no SICA was omitted because they all support one of the selected IaC tools in chapter 4.2. Next, the built-in tools (e.g., *terraform validate*) were removed so that only third-party tools remained. Thereby, six tools were omitted (*terraform validate*, *puppet parser validate*, *helm lint*, *ansible --syntax-check*, *terraform fmt*, *helm template*). The built-in SICAs were still incorporated into the *IaC Analyzer Decision Guide* in chapter 4.6 but were not subject to the assessment in chapter 4.5.

Next, it made no sense to assess meta-linters, which incorporate several other SICAs without adding additional features. Even though these tools are incorporated into the *IaC Analyzer Decision Guide*, they were not considered for an in-depth assessment. Hence, six tools were omitted (*super-linter*, *mega-linter, molecule, ValidKube, ValidIaC,* and *trunk*). To this point, twelve tools were excluded, leaving 32 on the list.

In the following step, all SICAs that do not have a GitHub repository (but are still free to use or have a freemium model) were excluded. As with the criteria before, these tools are still incorporated into the *IaC Analyzer Decision Guide* but not used for further assessment. This criterion was applied because the assessment findings (e.g., high false positive rate) shall be easy to fix. This is only possible for tools where researchers and practitioners have access to the source code. Still, since all repositories of the remaining tools in the list are accessible, no tools were excluded.

Next, all tools that do not provide any outcome that can be assessed were excluded. For instance, *infracost* established a cost estimate by investigating *Terraform* resource files. Albeit, the outcome of this SICA cannot be evaluated since the findings do not imply misconfiguration or similar. Thereby, two more tools were removed (*infracost* and *inframap*).

In the next step, all tools without a standard set of rules were excluded from the assessment because custom rules would have to be created to assess the false positive rate. Thereby three tools were excluded (*conftest*, *OPA*, *shisho*). In total, 27 tools remain on the list.

Last, scoring was performed to select the **SICAs** for the practitioner assessment. As in chapter 4.2, the number of votes indicated popularity. The top ten SICAs with five or more votes were selected. Also, for each IaC tool that the selected SICAs did not yet cover, the top SICA for this IaC tool was selected, although the number of votes was below five. In exchange, the least popular tool that was on the list and covered by another SICA was removed. Table 19 summarizes the selected SICAs and the IaC tools they support.

Table 19. The Selected SICAs for an In-Depth Assessment

Name	**IaC Tool**	**Votes**
checkov	Multiple	41
KICS	Multiple	13
tfsec	Terraform	29
terrascan	Terraform	20
ansible-lint	Ansible	11
hadolint	Docker	7
cfn_nag	CloudFormation	7
KubeLinter	Kubernetes	4
puppet-lint	Puppet	5
Cookstyle	Chef	2

Source: Own representation.

4.4. Identification of Open-Source and Proprietary Infrastructure Repositories

According to the previously selected IaC tools and corresponding SICAs, open-source and proprietary repositories must be identified for the following IaC tools: *Terraform*, *Ansible*, *Docker*, *CloudFormation*, *Kubernetes*, *Puppet*, and *Chef*. Although the SICAs from Table 19, which cover multiple languages, could be used to scan the code of other IaC tools, too, this work is limited to the mentioned IaC tools.

4.4.1. Open-Source Repositories

Open-source repositories were identified in three different ways. In the first step, the author searched for **vulnerable-by-design repositories** for the specified IaC tools. For instance, *terragoat* (bridgecrew, 2020c) is a vulnerable-by-design repository with many *Terraform* misconfigurations. A maximum of one vulnerable-by-design repository per IaC tool was selected. How the remaining repositories were identified depended on whether the DSL of an IaC tool was recognized as its own programming language in GitHub (e.g., HCL for *Terraform*). Second, for IaC tools, whose DSL was recognized as its own programming language in GitHub, the **GitHub GraphQL API** was queried with the **RADON** tool (Dalla Palma et al., 2021). The results were filtered according to the inclusion and exclusion criteria from chapter 3.4, and suitable repositories were selected. Third, for IaC tools, whose DSL was not recognized as its own programming language in GitHub (e.g., YAML for

Ansible), the author used the **GitHub UI** to search for **specific tags** (e.g., *Ansible*). The repositories were selected according to the inclusion and exclusion criteria from chapter 3.4

A mixture of real-world and intentionally vulnerable code is essential for the in-depth assessment of the SICAs. This is because it guarantees that the SICAs will find various misconfigurations via the intentionally vulnerable code. At the same time, it also shows how the SICAs perform with real-world code. The repositories presented in Table 20 were identified using vulnerable-by-design repositories, the GitHub GraphQL API, and the GitHub UI. It answers the research question of this chapter: *For each selected IaC tool, what are two suitable open-source repositories to conduct static code analysis?* The identified repositories are used for the assessment in chapter 4.5.2.

Table 20. Open-Source Repositories for the Selected IaC Tools

IaC Tool	Name	Source	Stars	Description
Terraform	*terragoat*	bridgecrew (2020c)	763	Vulnerable-by-design
	cloudblock	Geary (2020)	562	Ad-blocking application
Ansible	*postgresql_cluster*	Kukharik (2020)	582	Highly available PostgreSQL cluster
	openshift-ansible	OpenShift (2016)	2,070	OpenShift cluster configuration
Docker	*dockerfiles*	He (2014)	745	A collection of *Dockerfiles*
	Dockerfile	Sekhon (2016)	399	A collection of *Dockerfiles*
CloudFormation	*cfngoat*	bridgecrew (2020a)	72	Vulnerable-by-design
	aws-cf-templates	widdix (2015)	2,520	*CloudFormation* Templates
Kubernetes	*kubernetes-goat*	Akula (2020)	2,595	Vulnerable-by-design
	kubernetes-examples	Container Solutions (2017)	1,150	Self-contained *Kubernetes* examples
Puppet	*puppet-nginx*	Fryman (2014)	466	*Puppet* module for NGINX
	puppet-os-hardening	DevSec Hardening Framework Team (2018)	255	*Puppet* module for OS hardening
Chef	*aws*[9]	Sous Chefs (2012)	518	AWS Cookbook
	chef-cookbook	RabbitMQ (2012)	214	Cookbook for RabbitMQ

Source: Own representation.

[9] The repository does not have two commits per month on average. However, no more suitable repository could be identified.

4.4.2. Proprietary Repositories

The author contacted Capgemini employees he knew had access to repositories for one of the required IaC tools. Even though *Chef* and *Puppet* are popular configuration management tools, the author could not identify any *Chef* or *Puppet* repositories. Often, *Chef* and *Puppet* were replaced by *Ansible*. Similarly, no *CloudFormation* repositories could be identified because *Terraform* was used instead. As a result, no *CloudFormation*, *Chef*, and *Puppet* repositories were identified. Ultimately, the affected SICAs are not executed against proprietary repositories.

Even though *Chef*, *Puppet*, and *CloudFormation* were not covered with proprietary repositories, the author identified repositories for all other IaC tools. Table 21 overviews the selected proprietary repositories. It answers the research question of this chapter: *For each selected IaC tool, what are two suitable proprietary repositories to conduct static code analysis?*

Table 21. Proprietary Repositories for the Selected IaC Tools

IaC Tool	Description
Terraform	Provisioning of an application to configure specific products on AWS.
	Provisioning of an application to send notifications on AWS.
Ansible	Configuration of a VM running Prometheus and other internal services.
	Configuration of multiple VMs running a Kubernetes cluster.
Docker	A Dockerfile for a Go Prometheus exporter.
	A Dockerfile for an Angular web application.
~~CloudFormation~~	No repository could be identified.
	No repository could be identified.
Kubernetes	Orchestration of several internal services of an observability stack.
	Orchestration of several internal development services
~~Puppet~~	No repository could be identified.
	No repository could be identified.
~~Chef~~	No repository could be identified.
	No repository could be identified.

Source: Own representation.

4.5. Assessment of Static Infrastructure Code Analyzers

This chapter analyzes the characteristics of all 44 discovered SICAs, and an in-depth assessment of the findings of the ten selected SICAs. The SICAs' characteristics were extracted from the GitHub repositories, official documentation, and internet articles. Additionally, some characteristics were identified by installing and executing the SICAs. Capabilities of the SICAs that could not be identified with one of the above methods were not integrated into the assessment. Regardless of this, as will be shown in chapter 4.6, practitioners and researchers identifying missing information can easily integrate it in the future. In addition to analyzing the SICAs' characteristics with the above sources, the in-depth assessment is based on the previously identified repositories.

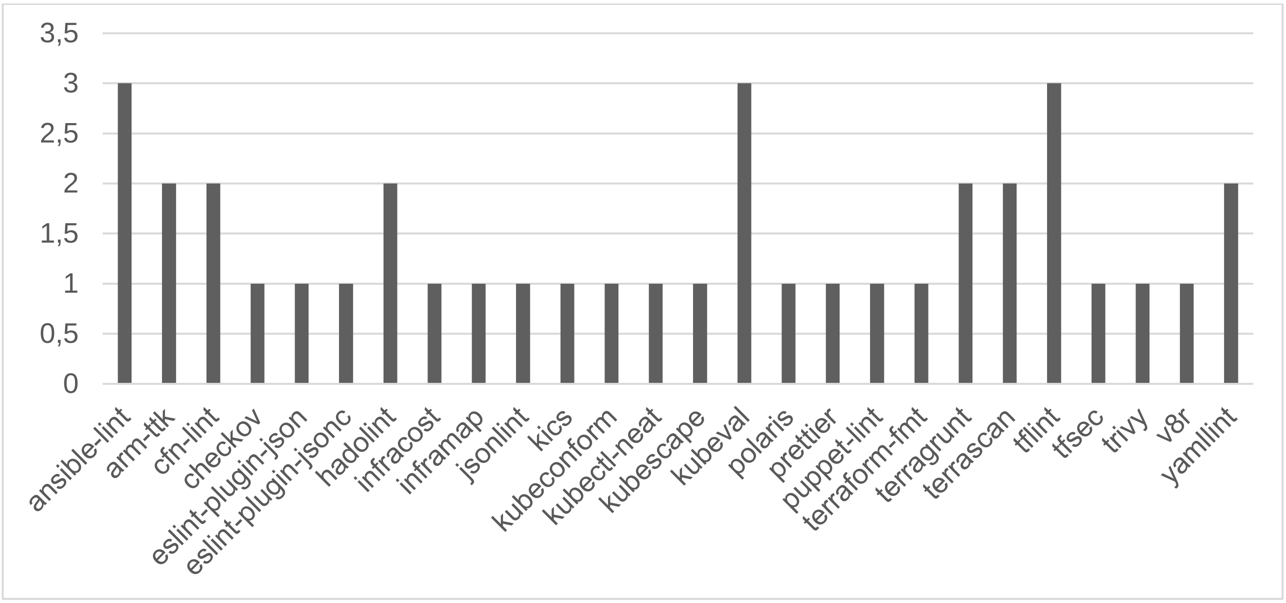
4.5.1. RQ 3.5.1: What are the Characteristics of the Identified SICAs?

The research question of this chapter is: *What are the characteristics of the identified SICAs?* It is progressively answered in the following paragraphs, each describing a different characteristic of the identified SICAs.

Meta-Tools

The meta-tools, which integrate several other SICAs, were only analyzed regarding the included SICAs. This is because the included SICAs are individually analyzed in-depth, and the meta-tools do not add functionality. Five of the 44 analyzed SICAs are meta-tools. Figure 14 overviews all SICAs integrated into the five meta-tools. SICAs for the most popular IaC tools, like *ansible-lint* for *Ansible*, *kubeval*[10] for Kubernetes, or *tflint* for *Terraform,* are integrated into most meta-tools.

Figure 14. SICA Characteristics: IaC Tools Included in the Analyzed Meta-Tools



Source: Own representation.

Built-In Tools

Five of the 44 analyzed tools are built-in tools (e.g., *terraform validate*). A built-in tool is built into the IaC tool, whereas third-party tools must be installed separately. *Terraform* has two built-in SICAs, whereas *Helm*, *Puppet*, and *Ansible* have one. Since none of these SICAs is open-source, they

[10] kubeval is not considered in this work because it was last updated longer than one year ago.

cannot be analyzed like third-party tools. Ultimately, these tools are excluded from the statistics discussed in the following.

Automatically Resolving Issues

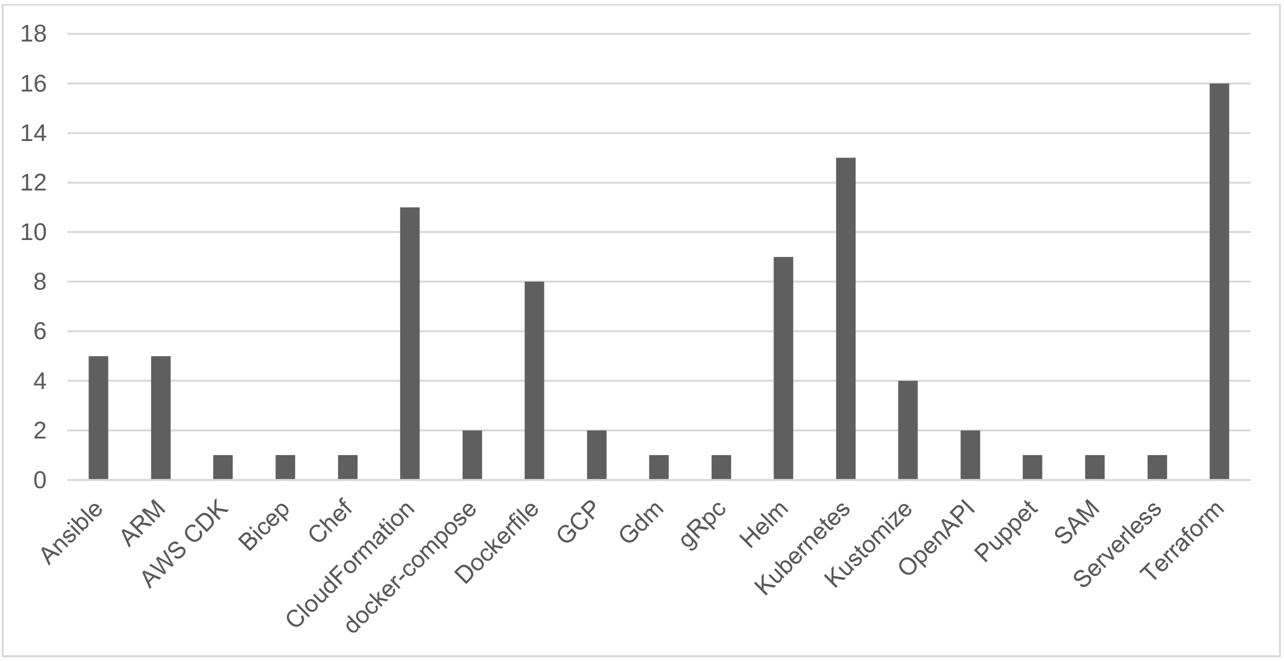
Six SICAs support automatically resolving issues. The small proportion is probably because many findings cannot be automatically resolved. Instead, a human needs to be involved in evaluating the findings. For example, a publicly accessible *S3 bucket* is often not intended. Regardless, in a rare case where this behavior is intended, practitioners do not want a SICA to automatically resolve this issue before they can add a comment to ignore the finding.

The following statistics are for the 34 SICAs that are **neither built-in tools nor meta-tools,** that is, all **third-party SICAs** that do not merely execute other SICAs.

IaC Tool and File Type Support

Either SICAs are specific to one or more IaC tools, or they are general-purpose and support one or more file types. The latter approach allows the user to create custom rules for any IaC tool that supports the particular file type. Figure 15 illustrates the IaC tool support among all analyzed SICAs. The diagram clearly shows that most SICAs support the most popular IaC tools: *Terraform* (provisioning), *Kubernetes* (orchestration), *Docker* (server templating), *Helm* (package manager), and *Ansible* (configuration management). These IaC tools had the most votes in their respective categories during the MLR.

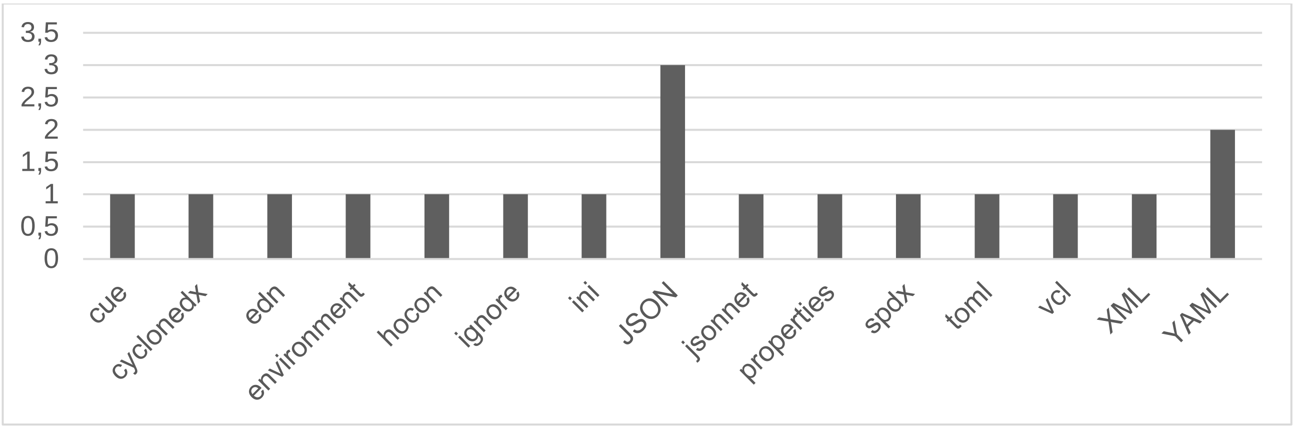
Figure 15. SICA Characteristics: IaC Tool Support



Source: Own representation.

Beyond analyzing the supported IaC tools, it is also interesting to see which file types general-purpose policy engines like *Rego* support. This is important because not all IaC tools are covered by SICAs. For instance, none of the selected SICAs supports *Docker Compose* natively. Hence, a general-purpose policy engine supporting YAML files is necessary to create custom policies for a *Docker Compose* file. As a result of the SICA analysis, Figure 16 shows that the analyzed SICAs support the file types YAML (2) and JSON (3) the most.

Figure 16. SICA Characteristics: File Support



Source: Own representation.

Development Support

The analysis of the development support is divided into three categories: IDEs, CI platforms, and VC. First, VS Code (13) is the most supported IDE, followed by Jetbrains (2). Next, most SICAs mentioned GitHub Actions (20), followed by GitLab CI (9) and Jenkins (8) as their supported CI platforms. The data about CI platforms include those explicitly mentioned in the documentation of the SICAs. Still, using the SICAs in other CI platforms might be possible. Last, the only VC integration mentioned in the SICAs' documentation was pre-commit-hooks (5).

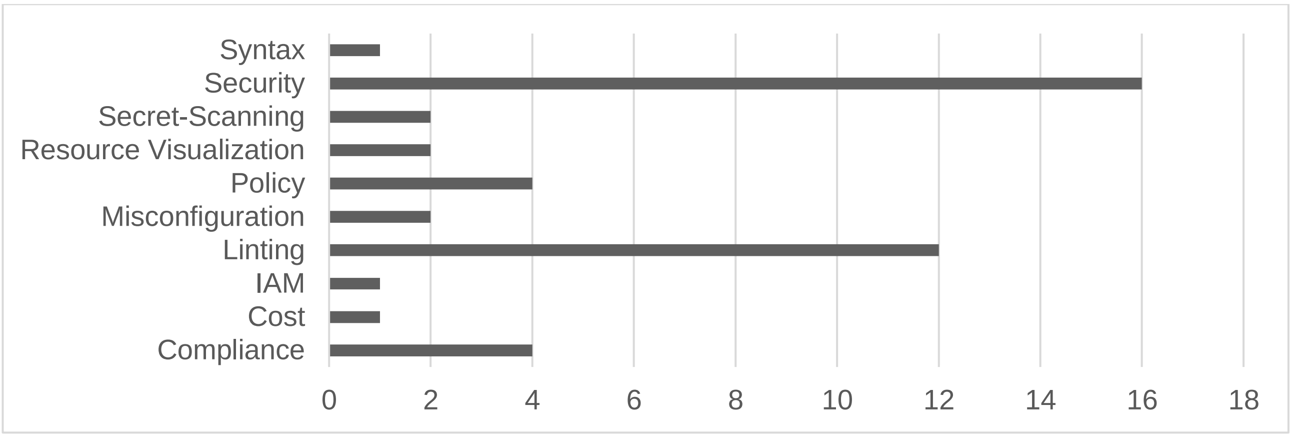
Rules

In total, 44 SICAs were analyzed, and 30 have built-in rules. Those without built-in rules were general-purpose tools or not opinionated and left all of the rule creation to the user. Furthermore, 27 tools allow custom checks, whereas the users are restricted to the built-in rules or their own rules for the other 17 tools. For 23 tools, rules can be explicitly blacklisted. Consequently, these rules are not executed during the scan. Twenty-two tools allow whitelisting rules. Whitelisting means explicitly configuring which rules the SICA should execute instead of blacklisting all unwanted ones. Nineteen SICAs allow ignoring specific findings, most often by adding a comment to the infrastructure manifest stating the rule to ignore and explaining why to do so. The high numbers show how important it is in practice to be able to customize a SICA to the specific environment in which it is used.

Categories

Classifying the SICAs into one or more categories is not straightforward. To simplify the classification, the author decided to unite the categories of best practices and anti-patterns under the term linting. Another challenge was that the categories overlap. For instance, a custom compliance rule can simultaneously be a security rule. A SICA can moreover belong to more than one category. With this in mind, the author mainly reused the categories mentioned in the SICAs' documentation. With that procedure, security and linting were the categories the most tools belonged to (16 and 12, respectively). The second and third most common categories were compliance and policy (each 4). Moreover, there were only one cost estimation tool and two resource visualization tools. Beyond that, one SICA analyzes AWS IAM configurations. Figure 17 illustrates how many SICAs belong to each category.

Figure 17. SICA Characteristics: Categories



Source: Own representation.

Output

Analyzing the output options of the SICAs is not ambiguous since the tool vendors use different descriptions for the outputs, and no standard can be used to unify the terms. Even though many different formats are used, some output formats stand out. JSON is supported by 27 SICAs, sarif by 12, and JUnit by nine. Hence, JSON is the most supported output format, and researchers should consider using it for their SICAs to align with the standard. The author added the output type *text* to those SICAs that allow users to specify a particular output format and have no name for the default format. For instance, *salt-lint* (Warpnet B.V., 2020) offers JSON output via the *--json* command line option. Since the default output format has no name, the author added the value *text* to salt-lint's output formats. Beyond that, many other output types were mentioned, however, often only once. Appendix E gives an overview of all output formats mentioned.

Backers

Only six of the 44 analyzed SICAs are backed only by the community. Companies back all other 38 tools. Often, these companies offer additional commercial support and tool features (e.g., *checkov*). Generally, these tools are more actively maintained than community projects.

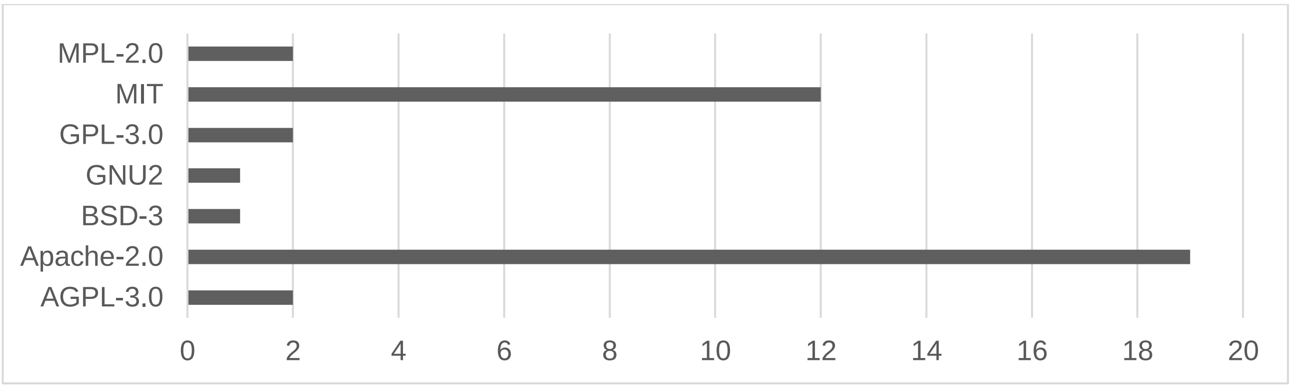
GitHub Statistics

When added together, the identified SICAs have over 95,000 GitHub stars. On average, each tool has more than 2,850 stars. This observation illustrates the popularity of these SICAs.

License

Figure 18 illustrates the licenses used by the SICAs. Importantly, every SICA is licensed. Otherwise, usage of the SICA without explicit permission would be forbidden (Häußge, n.d.). Apache-2.0 (19) and MIT (12) are the two licenses used by most tools. According to Häußge (n.d.), Apache-2.0 and MIT licenses "require that attribution must be kept in place, warranty & liability are strictly ruled out and … commercial use is no problem" (Häußge, n.d.). In essence, if practitioners or researchers create new SICAs, they might consider the Apache-2.0 or MIT licenses. A comprehensive comparison of the open-source licenses is provided by Rathee and Chobe (2022).

Figure 18. SICA Characteristics: Licenses



Source: Own representation.

Web Applications

Seven of the identified SICAs offer web applications that practitioners can use to test their infrastructure code without installing a tool (see Table 22). Such web applications are beneficial if only a small number of infrastructure manifests must be tested, like one single *Dockerfile,* or if the SICA is only evaluated for further usage. Practitioners and researchers developing new SICAs should consider offering their tool via a web application from the beginning to allow easy evaluation of the tool. However, users of these web applications must be careful if uploading their code (i.e., using the web application) is permitted by law. Some tools can only be used via such a web application. For instance, *ValidKube* can only be accessed with a web browser. Of the seven tools listed in Table 22, two are meta tools (*ValidIaC*, *ValidKube*), four support one or more IaC tools (*hadolint*, *kube-score*, *semgrep*, *shisho*), and *OPA's* general-purpose language *Rego* supports JSON files. This finding suggests that such web applications can be developed for all kinds of SICAs.

Table 22. SICA Characteristics: SICAs with Web Applications

Tool	**URL**
hadolint	https://hadolint.github.io/hadolint/
kube-score	https://kube-score.com/
OPA	https://play.openpolicyagent.org/
semgrep	https://semgrep.dev/playground/new
shisho	https://play.shisho.dev/
ValidIaC	https://validiac.com/
ValidKube	https://validkube.com/

Source: Own representation.

4.5.2. RQ 3.5.2: How do Practitioners Assess the Usefulness of the Findings of the Identified SICAs?

This chapter elaborates on the quality of the findings of the SICAs selected for the in-depth assessment. It answers the research question: *How do practitioners assess the usefulness of the findings of the identified SICAs?* Before analyzing the assessment results, it is vital to notice that this assessment does not allow a comparison of the tools. This is because the tools are executed against repositories of different sizes and types. Furthermore, the assessment cannot be directly used for a statement about the quality of a SICA since most SICAs allow disabling specific rules. Hence, only because a tool has a high false positive rate does not mean it is impractical to use. It is necessary to fine-tune the SICAs for each repository. In the same way, this is necessary for static analyzers for general-purpose programming languages, for instance, SonarQube for Java. The goal of the assessment is only to get an impression of the unconfigured SICAs. Ultimately, it is more important whether a SICA allows to blacklist rules or ignore specific findings than to have a perfect true positive rate right from the start, as analyzed in chapter 4.5.1.

Some SICAs could have been used to test more than one IaC tool. For instance, *checkov* could have been used to test *CloudFormation* and *Terraform* code. Nevertheless, this approach was not taken because such a complex analysis is out of the scope of this work. Besides, *CloudFormation*, *Chef*, and *Puppet* repositories could not be identified at Capgemini. Hence, the results of *cfn_nag*, *cookstyle*, and *puppet-lint* refer only to the analyzed open-source repositories. The detailed findings are listed in Appendix F, and Table 23 overviews the assessment result. Legend: CKS = Cohen’s Kappa Score, FP = False Positive, FIX = How many findings would practitioners resolve? The Cohen’s Kappa score is calculated for each repository. Hence, depending on how many repositories have been tested, the column contains two to four values.

Table 23. SICA Characteristics: Cohen’s Kappa Score, False Positive Rate, Fix Rate, and Execution Time of the Selected SICAs

SICA Name	CKS	FP	FIX	Time
checkov	0.059, -0.036, 0, 0.164	17%	68%	11 s
KICS	0, 1, -0.066, -0.063	15%	65%	72 s
tfsec	-0.042, 1, -0.079, 0	18%	51%	8 s
terrascan	1, 1, 0, 0.375	12%	62%	9 s
ansible-lint	1, 0, 0.837, -0.149	27%	58%	18 s
hadolint	0, 0, 1, -0.8	16%	85%	2 s
cfn_nag	1, 1	0%	54%	3 s
KubeLinter	1, 1, 0.291, 0.087	15%	85%	30 s
puppet-lint	1, 1	0%	52%	2 s
Cookstyle	1, 1	0%	81%	4 s

Source: Own representation.

Overall, practitioners considered most findings (on average 88%) as true positives, agreeing with the findings. Hence, the rules of the analyzed SICAs seem to be widely accepted in practice. Nevertheless, practitioners would resolve significantly fewer findings (on average 66%), although they agree to these rules. Both observations underline the importance of having the option to disable rules and ignore specific findings in SICAs. Developers might want to ignore findings because a rule does not apply to a particular scenario or requires too much effort to implement. Supporting this assumption, one practitioner explained that he wanted to ignore specific findings with comments. This was because he generally agreed to a rule but identified exceptions to the rule. Furthermore, the agreement levels between the two raters measured by Cohen's Kappa score vary significantly. The raters agreed almost perfectly (CKS 0.81 - 1.00) in 15 instances, whereas the agreement was fair (CKS 0.21 - 0.40) in 2, slight (CKS 0.00 - 0.20) in 10, and poor (CKS < 0.00) in 7 cases (Landis & Koch, 1977, p. 165). This finding implies that evaluating the SICAs' findings is highly subjective.

Beyond that, the analysis showed that scan times are fast, often below 30 seconds. Hence, if SICAs are integrated into a CI/CD pipeline, the increased execution time of the pipeline is negligible.

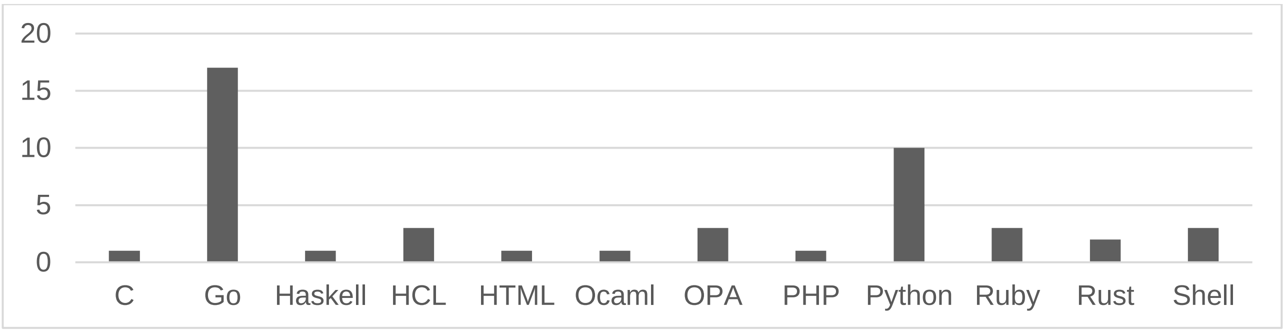
4.5.3. RQ 3.5.3: How are the Identified SICAs Implemented?

Chapter 4.5.1 explained the characteristics of the discovered SICAs. In contrast, this chapter elaborates on the implementation of the SICAs. It answers the research question of this chapter: *How are the identified SICAs implemented?* The findings suggest how new SICAs could be implemented. Practitioners and researchers could use the same approaches to take advantage of the benefits others exploited or very different approaches to prove that these are also feasible.

Programming Languages

The analysis of the implementation details (programming languages and rule implementation) of the SICAs revealed that most tools are either implemented in Go or Python. For future work, it could be interesting to investigate the exact implementation details of the analyzed SICAs and the practitioners' motivation for choosing the particular programming languages. Notably, HCL is used for the implementation of tests, not the implementation of the SICAs. To conclude, Figure 19 illustrates the programming languages in which the identified SICAs are implemented and shows that Go (17) has been used the most, followed by Python (10).

Figure 19. SICA Characteristics: Programming Languages for the Tool Implementation

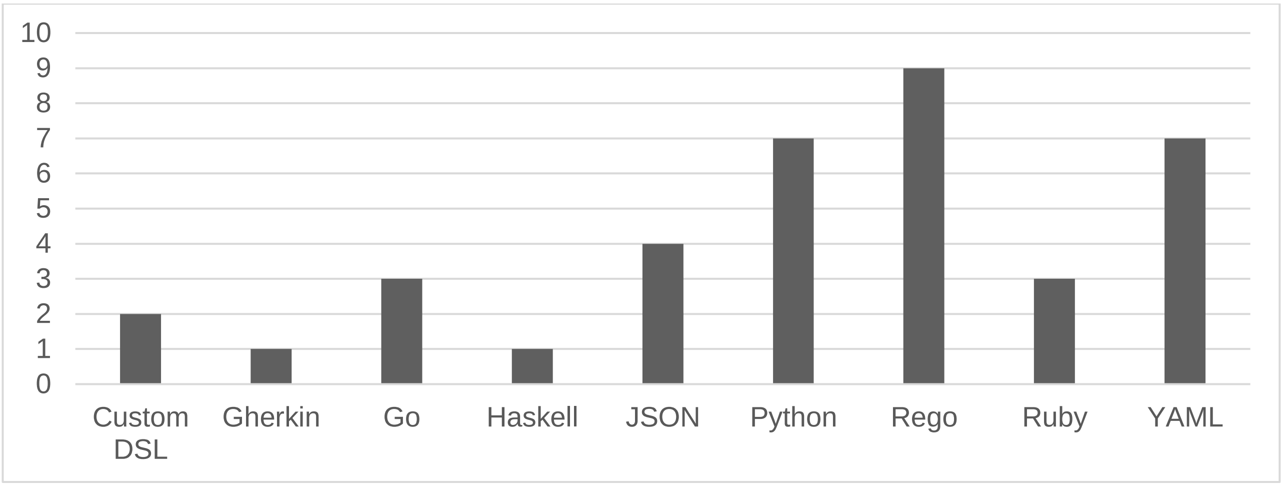


Source: Own representation.

Rule Languages

Most SICAs' rules are implemented either with JSON (11%) or YAML (19%) files, with general-purpose programming languages like Go (8%) or Python (19%), or using a general-purpose policy engine like *Rego* (24%). Some tools also offer a mixture of these approaches, like *checkov*, which supports custom rules in Python or YAML, and *terrascan,* which implements each rule with a JSON and a *Rego* file. The advantage of implementing rules in JSON or YAML is that custom rules can be easily created without prior knowledge of a particular programming language. Nevertheless, rules written in general-purpose programming languages may be more expressive and allow more complex rules. Policy languages, like *Rego*, are easy to use and expressive at the same time. Figure 20 shows that YAML, *Rego*, Python, and JSON are the four most used rule languages.

Figure 20. SICA Characteristics: Languages Used for the Rule Implementation



Source: Own representation.

Since there already are general-purpose policy engines like *Rego*, it is questionable whether it makes sense for researchers to implement their own custom DSLs when creating new SICAs. Instead, they could use one of the existing ones, like *Rego*. If their focus is a new SICA, they can rely on existing policy engines and spend most of their time developing new rules instead of dealing with the implementation details of such a rule engine. The ease of implementing such a rule in *Rego* is shown in Listing 3. It shows a sample *Rego* rule that checks whether an *AWS EBS* volume is encrypted.

Listing 3. Example of a Rule written in Rego

```
package rules.simple_rule_custom_message
resource_type = "aws_ebs_volume"

deny[msg] {
  not input.encrypted
  msg = "EBS volumes should be encrypted"
}
```

Source: Regula, 2022.

In contrast to all approaches discussed so far, *terraform-compliance* takes an entirely new approach by allowing users to write behavior-driven development (BDD) rules with Gherkin. An example of such a BDD rule is shown in Listing 4. It exemplifies the expressiveness of BDD rules for an elementary example. Still, it would be interesting to analyze how expressive the format is for complex rules, including nested structures or similar.

Listing 4. Example of a Behavior-Driven Development Rule for terraform-compliance

```
Feature: All resources must have specific tags
Scenario: Ensure all resources have the whitelist tag
  Given I have resource that supports tags defined
  Then it must contain tags
  And its value must contain whitelist
```

Source: terraform-compliance, n.d.

4.5.4. RQ 3.5.4: Which IaC Tools and Use-Cases Lack SICAs?

This chapter elaborates on current areas in which SICAs are still missing. These areas offer attractive possibilities for researchers and practitioners to implement such tools. In addition to the previous MLR on SICAs, the author specifically searched for tools of a particular type and IaC tool before expressing in the matrix (Table 24) that no such tools exist. Thereby, additional SICAs were identified[11]: *salt-lint* (Warpnet B.V., 2020) and *docker-compose-viz* (Group PSIH, 2020). Also, the tools supporting multiple IaC tools had to be investigated in detail. For instance, *KICS* states to be a security SICA and to support, among other IaC tools, *Ansible*. The source code and documentation had to be analyzed to determine whether KICS has security rules for *Ansible* and not only for the other IaC tools. Besides, although an area for a particular IaC tool may be indicated as covered in Table 24, it does not mean it is fully covered. There is always room for improvements and new rules. Nevertheless, creating an entirely new SICA from scratch may not be necessary for these areas.

The matrix illustrated in Table 24 answers **RQ 3.5.4** by showing which gaps exist. These gaps are suitable for the future work of practitioners and researchers. It shows the coverage of the analyzed SICAs regarding the defined categories. All cells with a dash indicate areas for future research. There are scenarios where a specific combination of a category and an IaC tool makes no sense. For instance, *Ansible* manifests cannot be used to estimate the costs since the costs depend on where the VMs are deployed. These scenarios making no sense are indicated with NA (not applicable).

The matrix shows that SICA coverage for well-established IaC tools like *Ansible*, *Terraform*, *Kubernetes*, and *Docker* is excellent, whereas less popular IaC tools like *Nomad* are understudied. For some tools, it is more challenging to create SICAs. For instance, *Pulumi* supports various general-purpose programming languages. New solutions must be designed for a SICA for *Pulumi*. For instance, one SICA per programming language could be implemented. Alternatively, a parser

[11] The identified SICAs were incorporated into chapter 4.3.

would be required to translate the different programming languages into an intermediate format first, which could then be interpreted by another analyzer. In an approach where one SICA per general-purpose programming language is created, a static analyzer could check for the presence of specific values. For instance, Listing 5 shows the *Pulumi* configuration for an *AWS S3* bucket in TypeScript. In line four, *s3.BlockPublicAccess.BLOCK_ALL* specifies that, as the name tells, all public access shall be blocked. Similar to other SICAs for *Terraform AWS* or *CloudFormation*, a static analyzer could give the user a hint that, if they allow public access, their bucket can be accessed by everyone. Of course, this could be the desired behavior. A comment in the fourth line of Listing 5 could silence the SICA for the specific finding, as in many other SICAs.

Listing 5. S3 Bucket Defined with Pulumi in Typescript

```
const bucket = new s3.Bucket(this, 'example-bucket', {
    accessControl: s3.BucketAccessControl.BUCKET_OWNER_FULL_CONTROL,
    encryption: s3.BucketEncryption.S3_MANAGED,
    blockPublicAccess: s3.BlockPublicAccess.BLOCK_ALL,
});
```

Source: Pulumi, 2022.

Beyond that, SICAs visualizing the deployed infrastructure only exist for *Terraform* (*inframap)*. Hence, developing such SICAs for other IaC tools, especially provisioning tools, would be interesting. Similarly, *infracost* is the only tool that estimates the cloud costs and works only with *Terraform*, particularly with the most popular public cloud providers, AWS, Azure, and Google Cloud. Estimating the costs most often involves communication with a cloud API to retrieve pricing information which conflicts with the view of static analysis in this work. Still, cloud cost estimation tools for other provisioning tools or smaller public cloud providers could be a topic for future research.

To sum it up, the matrix illustrated in Table 24 answers **RQ 3.5.4:** *Which IaC Tools and Use-Cases Lack SICAs?* It illustrates suitable gaps for future work by practitioners and researchers indicated with dashes. Combinations of IaC tools and categories making no sense are indicated with NA.

Table 24. How Existing SICAs Cover the Use Cases Visualization, Cost, Misconfiguration, and Security for the Selected IaC Tools

IaC Tool	**Visualization**	**Cost**	**Misconfiguration, Compliance, Linting, Policy, Syntax**	**Security**
Ansible	NA	NA	x	x
Puppet	NA	NA	x	[12]
Chef	NA	NA	x	-
Salt	NA	NA	x	-
Terraform	x	x	x	x
CloudFormation	[12]	-	x	x
ARM	-	-	x	x
Pulumi	-	-	-	-
Kubernetes	[13]	NA	x	x
Docker Swarm	-	NA	-	-
Nomad	-	NA	-	-
Docker Compose	x	NA	x	-
Docker	NA	NA	x	x
Vagrant	NA	NA	x	-
Packer	NA	NA	x	-
Helm	-	NA	x	x

Source: Own representation.

4.6. Design and Implementation of the Static IaC Analyzer Decision Guide

A significant challenge of this work is sharing the collected data with practitioners and researchers. The data was gathered in the two multivocal literature reviews and the assessment of the SICAs with practitioners. At this point, it is only available in an Excel sheet. Summarizing all characteristics of all identified SICAs is hardly possible within the text part of this thesis. Also, it is rather unlikely that practitioners and researchers will discover this thesis. A blog article or a publicly available Excel spreadsheet would be an easier way to share the knowledge. However, keeping such a source up to date is challenging since communication between the community and the author of this work would mainly happen in the blog post's comments or the Excel spreadsheet's comments. Beyond that, an article or Excel sheet would require practitioners to identify SICAs for their particular use case manually. Hence, it was decided to create an interactive application encoding the data gathered in the previous chapters. After all, this chapter describes the design and development of the *IaC*

[12] There is a tool but it does not meet the requirements of this work (e.g., not enough stars on GitHub).
[13] There is a tool but it is not pure static analysis.

Analyzer Decision Guide in three cycles, each consisting of four phases: analysis, design, evaluation, and diffusion.

4.6.1. Analysis

This chapter answers the research question **RQ 3.6.1**: *What is the current problem with finding suitable static code analyzers for an infrastructure project?*

First Cycle

Österle (2010, p. 13) explicitly mentioned that a DSR problem must be relevant. A relevant problem could be information overflow, for instance. The MLR on SICAs has demonstrated that such an information overflow challenge is given, considering the large number of sources and SICAs. At least some internet articles compared existing SICAs, shortly describing their functionality. However, practitioners may not have the time to read through these articles. Instead, they want a quick recommendation. Also, these articles often only describe the SICAs on a fundamental level. Beyond that, researchers need to determine which SICAs exist to direct future research. These observations suggest that the **practical problem** of developing a supporting tool for researchers and practitioners to find suitable SICAs is given.

Furthermore, Morris (2020) described a lack of SICAs. Chapter 4.3 showed that while such tools exist, identifying them requires considerable time. Also, practitioners must evaluate whether those tools match their project's requirements. This indicates a **gap in research and practice**, as outlined by researchers (A. Rahman, 2018a; A. Rahman, Elder, et al., 2018) and practitioners (Brikman, 2022; Morris, 2020). The **research question**, an integral element of each DSR project, has already been defined in chapter 3.6.1 (Österle, 2010, p. 4).

The **plan to develop the system** was to evolve from a basic implementation to a more complex one within three iterations, as suggested by Knauss (2021). The **scope** of the developed application was limited to static infrastructure code tools. However, it could be extended to include other tools, like container scanning tools. Beyond that, Knauss (2021) suggested defining the artifact early. Hence, the **artifact** was defined as a software application that helps practitioners and researchers choose one or more SICAs based on the requirements they enter into the application. This application will be referred to as the artifact or *IaC Analyzer Decision Guide* in the remainder of this chapter. Last, the **requirements** for the *IaC Analyzer Decision Guide* were established. The requirements were created according to the guidelines of Koelsch (2016). In the list of requirements below, the *IaC Analyzer Decision Guide* is abbreviated as *IACA-DG*. First, the functional requirements are described. Functional requirements define the functions of a system, whereas non-functional requirements describe the system's behavior and constraints (Koelsch, 2016, chapter 2).

Functional Requirements

- 1-1 The IACA-DG shall be accessible by everyone without registration or login.
- 1-2 The IACA-DG shall show all available options to the user.

- 1-3[14] The IACA-DG shall allow the user to rate the level of influence of the selected options.
- 1-4 The IACA-DG shall return all SICAs that match the user's selected options.
- 1-5 The IACA-DG shall return all available SICAs when a user does not select any option.
- 1-6[14] The IACA-DG shall calculate a score for each SICA it returns based on the selected options.
- 1-7 The IACA-DG shall hide the details of the returned SICAs from the user by default.
- 1-8 The IACA-DG shall show the details for a particular SICA when the user expands the detail section.

Non-Functional Requirements

- 2-1 The IACA-DG shall comprise a frontend application written in Typescript with React and a backend application written in Go with Gin.
- 2-2 The IACA-DG shall support the definition of SICAs in YAML only.
- 2-3 The IACA-DG shall have all its documentation on GitHub.
- 2-4 The IACA-DG shall roll back in case of a failed deployment.
- 2-5 The IACA-DG backend filter function shall return the results within 100 milliseconds 90% of the time.
- 2-6 The data of the IACA-DG shall stay up-to-date automatically.
- 2-7 The data of the IACA-DG shall be easily maintainable by everyone.
- 2-8 The data of the IACA-DG shall be protected against manipulation.
- 2-9 The IACA-DG shall be extendable to other types of IaC scanners.
- 2-10 The IACA-DG shall be maintainable by more than one person.
- 2-11 The IACA-DG shall not be responsive because it is unlikely that practitioners will search for SICAs on their phones or tablets.
- 2-12 The IACA-DG shall be deployed to production within ten minutes on each push to the main branch.

Second Cycle

The practical problem, research gap, research objectives, and research question remained the same as defined in the first cycle. What changed in contrast to the first cycle was the **plan to develop** the artifact and mainly the **scope**, most of which will be elaborated in the design phase. It was planned to let users rate how important specific options are to them in the first cycle of the design phase. Based on this rating, it was planned to calculate a score. However, the author recognized that users might feel overwhelmed by the number of options. This finding suggested omitting such an advanced scoring system. Therefore, *REQ 1-3* and *REQ 1-6* were excluded.

[14] This requirement was later omitted, see the analysis phase in the second cycle.

Third Cycle

After the second iteration, the author recognized that adding all available options to the UI leads to an overloaded and complex UI. Because of that, users could feel overwhelmed in the same way they would feel overwhelmed by having additional rating options. Also, users may tend to select too many options and therefore find no suitable tool. Hence, the scope of the application was adapted to include only the most decisive options on the UI. In this course, IaC categories were neither included on the UI. This is because they are highly biased by the tool vendors who categorize their tools and the practitioners' interpretation of these categories. The categories are also ambiguous. Is a policy blocking public access to an *AWS S3 bucket* a security measure simultaneously? In brief, IaC categories on the UI would have only confused users and distorted the search results.

4.6.2. Design

The analysis phase elaborated on the current problem with finding SICAs. This chapter designs a solution to this problem by answering the following research question: **RQ 3.6.2**: *How can the problem of finding suitable static code analyzers be overcome?*

First Cycle

The application's features are evident viewed from the user's perspective. A user selects several available options (*REQ 1-2*), and the application shows all SICAs that meet those criteria (*REQ 1-3*). A rough screen layout would illustrate the options at the top and the search results below. During the research about existing SICAs, the author discovered *ValidKube* (Komodor, 2022) and *ValidIaC* (Firefly, 2022), two web applications used to check infrastructure manifests (*Kubernetes* and *Terraform,* respectively) in the browser. *ValidIaC* is a fork of *ValidKube*. These web applications have an input section on the left and an output section on the right. Both applications are written in Go with Gin (backend) and Typescript with React (frontend) and deployed on AWS. The author decided to design the *IaC Analyzer Decision Guide* similar to these applications (REQ 2-1). Furthermore, in much the same way as *ValidIaC* and *ValidKube*, the *IaC Analyzer Decision Guide* allows public access without requiring users to log in or register (*REQ 1-1*).

Even though many aspects of *ValidIaC* and *ValidKube* can be adopted, several adjustments are required. For instance, there is no input section on the left and an output section on the right. Instead, the *IaC Analyzer Decision Guide* was designed to have an options section at the top (e.g., with a list of all IaC tools) (*REQ 1-2*) and an output section at the bottom where the matching SICAs are displayed (*REQ 1-4*). The tech stack of the *IaC Analyzer Decision Guide* is the same as *ValidKube* and *ValidIaC,* using Go with Gin for the backend and Typescript with React for the frontend. Still, the backend application of the *IaC Analyzer Decision Guide* pursues a different goal than *ValidKube* and *ValidIaC*. The main tasks of *ValidIaC's* and *ValidKube's* backend are calling the included tools (e.g., *tfsec*), parsing the outputs, and displaying the output on the UI. In contrast, the *IaC Analyzer Decision Guide* searches SICAs and filters them.

The first implementation question was **how and where the data about the SICAs were stored**. According to the requirements defined in the analysis phase, the data must be accessible by everyone so that it can be easily kept up to date (*REQ 2-7*, *REQ 2-10*). At the same time, the data must be resistant to manipulation, for example, from tool vendors who want to press their tools ahead (*REQ 2-8*). Due to those requirements, a database is not suitable to store the data because it would require an access control mechanism to decide who can update the data. Instead, the data are stored in YAML files in the project's GitHub repository as stated in the requirements (*REQ 2-2*, *REQ 2-3*). With that procedure, everyone can access the data by forking the repository, updating the data (adding, editing, or removing SICAs), and creating a pull request. The author of this work will then review the pull request to protect the *IaC Analyzer Decision Guide* against manipulation. The application was planned to be deployed on AWS with the already existing configuration of *ValidIaC* and *ValidKube* (*AWS Lambda* and *AWS S3*). Moreover, *ValidIaC* and *ValidKube* use Netlify (Netlify, n.d.) to deploy the React frontend. Since the author has no experience with Netlify, the static files of the React application were planned to be hosted via an *AWS S3 bucket*.

If a relational database was used to store the data about the SICAs, SQL could be used to **filter the data**. Since the data are stored in YAML files, another way had to be used to filter the SICAs based on the input criteria from the user. For such a scenario, the usage of a rule engine appears suitable based on the work of Andrzejewski et al. (2017). The authors built a decision guide for breast cancer treatment. Such decision guide systems implement rules to support experts in their decision-making. Based on these findings, the grule-rule-engine (Hyperjump, 2019) was used to implement the filtering feature.

As explained above, the data about the SICAs are stored in YAML files. All the characteristics identified during the analysis of the SICAs are stored in these files, one file per SICA. The schema for the YAML files is shown in Appendix G. Some of these characteristics can only be **updated manually**. For instance, if the documentation of a SICA is updated, the changes must be manually analyzed and transferred to the corresponding YAML file. At this point, automatically extracting the data is difficult. On the other hand, data that can be more easily accessed via an API, like the number of GitHub stars, can be **updated automatically** (*REQ 2-6*). This saves manual effort and ensures users maintain trust in the *IaC Analyzer Decision Guide* because its statistics are up-to-date. This observation suggested the development of a nightly job that retrieves the newest data about each SICA from the GitHub API, updates the YAML files, and pushes the changes to a branch intended for this purpose. After the changes have been merged to the main branch, a new application version is automatically deployed.

Second Cycle

In the second cycle, in addition to the design of the application itself, the **CI/CD** pipeline was defined (*REQ 2-12*). Since the repository is hosted on GitHub, GitHub Actions was used as CI/CD platform. First, the YAML definitions of the SICAs are checked against a schema, defined with the tool *yamale*

(23andMe, 2014). This schema check is the very first test in the pipeline, and if it fails, no further steps are executed. Second, static tests on the infrastructure code are performed. In this cycle, *hadolint* was introduced into the pipeline to check the *Dockerfile* for adherence to best practices. If these checks are successful, a Docker Image is built and pushed to DockerHub[15]. Lastly, the application is automatically deployed.

As mentioned in the analysis phase, it was decided not to allow users to **rate the importance of options**. Instead, a user's selection is binary. For instance, only tools that support the *custom rules* feature are displayed if users select the *custom rules* option. Otherwise, the user interface would become too complex. Hence, the code for the rating feature was removed, and *REQ 1-3* and *REQ 1-6* were omitted. In essence, the filtering process is straightforward. The *IaC Analyzer Decision Guide* filters the identified SICAs by the selected options. Empty options are ignored.

Third Cycle

ValidKube and *ValidIaC* have one *Dockerfile* for the Go application, whereas the React application is deployed via Netlify. The *IaC Analyzer Decision Guide* follows a different approach, as was decided in the third cycle. Both applications (**frontend and backend) are integrated into the same Docker Image**. The React application is served via the Go backend. Requests made to the path "/" are answered by Gin with the static React files, whereas all requests made to the paths "/api/*" are answered with the responses of the specific Go endpoints. In this way, deploying only one container reduces the monthly costs by 50%.

Furthermore, it was decided to **deploy the application to DigitalOcean**, not AWS, via *Terraform*. This is due to the author's experience with DigitalOcean's App Platform service. It is used to deploy the Docker Image containing the backend and frontend. The only resources needed to use the app platform are a Docker Image and the deployment configuration. The deployment configuration includes domain settings and resource limits (CPU and memory). When a deployment fails, DigitalOcean automatically rolls back to the last version, which worked adequately (*REQ 2-4*). The health check used by DigitalOcean to decide whether an application is healthy is configured in the *Terraform* manifest. Besides the app, *Terraform* deploys a domain resource and a project resource. The project acts as a namespace for all other deployed resources. The domain itself was retrieved from a domain name registrar, and the DNS settings point to the DNS servers of DigitalOcean, which allows DigitalOcean to connect the domain to the app.

Each time one or more commits are pushed to the main branch of the repository, the **GitHub Actions pipeline** is triggered, and a new version of the application is built and deployed (*REQ 2-12*). The GitHub Actions pipeline needed to be reworked to deploy the correct version of the Docker Image. On each push to the main branch, first, a new application version is automatically generated based on the commit history. The term "feat:" in a commit message indicates a minor version upgrade,

[15] The Docker Images are stored in the author's Docker Hub repository https://hub.docker.com/repository/docker/nileger/iac-analyzers

whereas “feat!” indicates a major version upgrade. All other commits are treated as patch version upgrades. Finally, the generated version is used to build a Docker Image and create a new GitHub release. Thereby, the built Docker Image is tagged with the generated version. This version number is then passed to *Terraform* as a variable to specify which Docker Image version must be deployed. Automatically generating the version number relieves developers of manually setting the version on each code update and possibly forgetting it.

Furthermore, the introduction of *Terraform* code into the repository suggested using static analysis to check the *Terraform* manifests for security issues. There are many *Terraform* SICAs that the user is invited to find using the *IaC Analyzer Decision Guide*. The author decided to use *tfsec* as a starting point, which may be exchanged or extended with other SICAs in the future. Hence, ***tfsec* was introduced into the CI/CD pipeline**. Because *tfsec* only had rules for the DigitalOcean resources *compute* and *spaces* (not for the resources *app*, *domain*, and *project*), one custom check was created by the author of this work. To elaborate on how convenient it is to create such a check, the author created a rule that checks whether a tag is specified for the Docker Image used in the app. Otherwise, a wrong, potentially outdated version could be deployed since, by default, the tag *latest* is selected. If the tag is removed from the *Terraform* manifest, an error is displayed by tfsec (Figure 21). Listing 6 shows the implemented check.

Figure 21. tfsec Output when the Custom Check Fails

```
Result #1 HIGH Custom check failed for resource digitalocean_app.sica-decision-
guide. The required image tag was missing
───────────────────────────────────────────────────────────────────────────────
  C:\<PATH>\iac-analyzers\terraform\app.tf:1-42
───────────────────────────────────────────────────────────────────────────────
    1  ┌ resource "digitalocean_app" "sica-decision-guide" {
    2  │   spec {
    3  │     name   = "sica-decision-guide"
    4  │     region = "fra1"
    5  │
    6  │     domain {
    7  │       name = var.domain_name
    8  │       type = "PRIMARY"
    9  └       zone = var.domain_name
   ..
───────────────────────────────────────────────────────────────────────────────
          ID custom-custom-cus001
      Impact By not having a tag, the latest tag will be taken by default which
may cause accidentally using a wrong version
  Resolution Add the image tag
  More Information
  - http://internal.acmecorp.com/standards/aws/tagging.html
───────────────────────────────────────────────────────────────────────────────
```

Source: Own representation (changed) based on Aqua Security Software, n.d.

Listing 6. Custom tfsec Check to Enforce the Configuration of a Docker Image Tag in the Terraform Manifest

```
checks:
- code: CUS001
  description: Check if the docker image referred in an app has a tag
specified
  impact: By not having a tag, the latest tag will be taken by default which
may cause accidentally using a wrong version
  resolution: Add the image tag
  requiredTypes:
  - resource
  requiredLabels:
  - digitalocean_app
  severity: ERROR
  matchSpec:
    name: spec
    action: isPresent
    value: spec
    subMatch:
      name: service
      action: isPresent
      value: service
      subMatch:
        name: image
        action: isPresent
        value: image
        subMatch:
          name: tag
          action: isPresent
  errorMessage: The required image tag was missing
  relatedLinks:
  - https://stevelasker.blog/2018/03/01/docker-tagging-best-practices-for-
tagging-and-versioning-docker-images/
```

Source: Own representation.

***Terraform Cloud* executes the *Terraform* commands** since it offers a free-to-use and easy integration into GitHub Actions. The GitHub Actions pipeline works differently based on how it is triggered. Whenever a developer creates a pull request to the main branch, the GitHub Actions pipeline executes the tests (*hadolint*, *tfsec*), builds but does not push the Docker Image, and runs the *Terraform plan* command but does not apply the changes. The output of the *Terraform plan* command is attached to the pull request. Thereby, the author of this work can investigate what would change if he accepted the pull request. The changes are merged to the main branch when he accepts the pull request. Then, the pipeline is triggered again, however, executing additional steps. In addition to the steps described before, the pipeline pushes the Docker Image to DockerHub and applies the *Terraform* code. Ultimately, a new version of the *IaC Analyzer Decision Guide* is deployed within ten minutes (*REQ 2-12*).

4.6.3. Evaluation

The analysis phase outlined the current problem with finding SICAs for IaC projects, and the design phase constructed a solution to this problem. How the constructed solution performs must be analyzed. For this reason, the evaluation phase assesses the artifact regarding the following research question: **RQ 3.6.3**: *How does the artifact support practitioners and researchers in finding existing SICAs?*

First Cycle

The evaluation in the first cycle was performed only by the author of this work because the application was not mature enough to be tested by others. The issues at this time were easy to be spotted. Beyond that, the author wanted to restrain the testers for later stages when their input was more relevant. At this point of development, the author recognized that the complexity introduced by the *grule-rule-engine* was high considering its benefit. This observation was undeniable when comparing the code to regular Go language constructs (i.e., if, else, for). Not to mention that only a few filter options (IaC tools, categories, and output) were implemented. Hence, the *grule-rule-engine* was replaced with regular Go language constructs in the second cycle.

Three **functional requirements** established in the analysis phase were targeted in this design cycle. First, the application was deployed in the public cloud without requiring users to register to use it (*REQ 1-1*). Second, the UI already showed a list of available options (*REQ 1-2*), and third, returned SICAs according to those options (*REQ 1-4*). Furthermore, six of the **non-functional requirements** were met (*REQ 2-1*, *2-2*, *2-7*, *2-8*, *2-10*, *2-11*). The applications were implemented in TypeScript (React) and Go (Gin) (*REQ 2-1*), and the SICAs were defined as YAML files (*REQ 2-2*). Since the code was published in a public GitHub repository, *REQ 2-7*, *2-8*, and *2-10* were also met. Also, the application was not responsive (*REQ 2-11*).

Ultimately, the foundation of the **research goal** and the solution to the **practical problem** was built in this cycle. Nevertheless, the *IaC Analyzer Decision Guide* was still immature and needed to be reworked in the following two cycles.

Second Cycle

As in the first cycle, the evaluation was performed only by the author for the same reasons. While testing the application, the author found that users have **no option to display all SICAs**. This finding suggested that users shall be allowed to display all SICAs by clicking the search button, which was decided to be implemented in the third cycle (*REQ 1-5*). Beyond that, SICA properties had to be adapted several times throughout development because some properties were still empty or wrong. Because the **YAML files are stored in folders named according to the IaC tool they support**, it was straightforward to find and update a particular SICA's characteristics.

In addition to the requirements fulfilled within the first iteration, the **functional requirements** *REQ 1-7* and *REQ 1-8* were met. The SICAs displayed on the UI were collapsed by default. When a user expanded a particular SICA, the detail section appeared. Four **non-functional requirements** were met. *REQ 2-12* was met by implementing the CI/CD pipeline. Since the application was deployed via DigitalOcean's App Platform, *REQ 2-4* was met since it supports automatic rollbacks in case of failure. The performance of the backend application was measured and revealed that the response times for the filtering endpoint were below 100 milliseconds 90% of the time, which met *REQ 2-5*. Furthermore, the architecture was reworked so that the *IaC Analyzer Decision Guide* could easily integrate other IaC scanners (*REQ 2-9*). With those improvements, the *IaC Analyzer Decision Guide* was one step closer to solving the **practical problem** and the **research goal**.

Third Cycle

The evaluation approach in the third cycle consisted of three types of tests. First, the application was checked against the requirements. Second, functional tests were performed to evaluate whether the tool returned the correct SICAs when users entered their criteria. In other words, the filtering process was tested. Third, the usability of the *IaC Analyzer Decision Guide* was elaborated on with usability test sessions. All tests were essential to ensure the benefit of the application.

The following **functional and non-functional requirements** were met in this cycle. The first functional requirement was met by allowing users to display all SICAs by selecting no options (*REQ 1-5*). Since *REQ 1-3* and *REQ 1-6* have been omitted, all functional requirements were met. Similarly, all non-functional requirements were met. First, the application was documented in the README.md file of the repository, thereby meeting *REQ 2-3*. Second and last, the script to automatically update the SICA data (e.g., number of GitHub stars) was implemented (*REQ 2-6*).

The **functional tests** were performed manually by the author of this work. Several use cases were constructed, and the expected outcomes were defined. These use cases were derived from the requirements of the first analysis cycle. The actual outcomes were then checked against the expected outcomes. These tests revealed a bug in the filtering process, which manifested in wrong outputs. Since this bug affected the main functionality of the tool, it was fixed after the third cycle. The bug occurred because the requirements were incorrectly sent to the backend. The functionality was once more tested after the fix had been applied. Another bug was identified by selecting all options, which did not return any SICAs. The application was supposed to display an info text explaining that users must change their requirements. However, as the test showed, the screen turned white. The bug was quickly identified, resolved, and tested.

Ultimately, three **usability test sessions** were performed. Although this was the last cycle, the findings of the usability test session were implemented if they were functional problems, critical problems, major problems, or minor problems. This was so that users could conveniently use the final version. Good ideas discovered during the usability test sessions will be implemented after the submission of this thesis. The usability test session plan describes the environment and setting of these sessions. It is shown in Appendix B. Appendix C shows the usability test script, which explains how the usability test sessions are performed in detail. The final report of the findings of the usability test sessions and their interpretation is in Appendix D. Still, the main findings of the test sessions are explained in the following. The three participants in the usability test sessions were one IaC expert, one experienced IaC developer, and one former researcher who now works for Capgemini as a business architect. The researcher is not an expert in IaC but computer science in general. All participants have at least some experience with IaC. Furthermore, they will be likely in the situation to decide on a SICA in the future. With this procedure, the target group was covered as much as possible.

Three main problems were identified during the usability test sessions. First, the usability test sessions revealed that the **filtering process was unintuitive** because participants were confused about how to display all available SICAs. Beyond that, they were confused about what would happen if they selected several options (i.e., whether the logical operator is “and” or “or”). Moreover, it was unintuitive for participants to have to scroll to reach the search button. Second, participants found it **challenging to locate data** in the detailed information sections of the SICAs. They found it confusing that the layout differed between SICAs. Furthermore, they expected a visualization of the SICA details beyond pure text. For example, participants suggested replacing boolean values with checkboxes. Third, participants wanted **descriptions of options** (e.g., ignore findings), **characteristics** (e.g., backers), and **how to use** the *IaC Analyzer Decision Guide* (e.g., logical operator).

These usability flaws were all resolved after the third cycle. First, the layout was changed to show the **filter options on the left and the output on the right**. An **info button** at the top of the filter section explains the applied logical operator and the meaning of filter options that might be unclear to users. Furthermore, **by default, all SICAs are displayed** in the output section. Moreover, the **search button was omitted**. Instead, a new search is triggered whenever the users change their selections. Beyond that, the **detail sections were restructured**. The content is divided by horizontal bars. The headings in all detail sections are the same, and if no information for a particular category is available, this is explained by a short information text. Furthermore, boolean values in the detail sections are represented as checkboxes instead of strings. Thereby, all identified usability flaws were resolved.

Beyond that, **two bugs** were identified. The SICAs were not sorted alphabetically, and the rule implementation filter did not work. These bugs were also resolved. Other participants' ideas are documented in Appendix D and will be analyzed and implemented after the submission of this thesis.

In summary, all requirements were met, and problems identified via functional tests and usability test sessions were resolved. Thereby, the **research goal** (finding a suitable format to publish the knowledge of this work) and **practical problem** (practitioners and researchers need a way to identify existing SICAs) were solved.

4.6.4. Diffusion

The problem with finding SICAs for IaC projects was described in the analysis phase. A solution to this problem was constructed in the design phase and assessed in the evaluation phase. With this in mind, the learnings of the development of the *IaC Analyzer Decision Guide* are explained and generalized. This answers the following research question: **RQ 3.6.4**: *What learnings can be applied to similar problems?*

First Cycle

According to Österle (2010, pp. 5-6), the developed artifact must be generally valid, which means it must apply to more than the particular problem. In the context of this work, this requirement suggests that the developed application must be easily transferrable to static application code analyzers, for instance. Moreover, it could also be transferred to use cases apart from static code analysis. The developed *IaC Analyzers Decision Guide* meets this requirement. Only the YAML files and code related to the characteristics of the tools would have to be adapted. It could also be extended to include more tools, for example, container scanning. This is the reason why *iac-analyzers.dev* has been chosen as the domain name. It allows the artifact to be extended in the future.

The following are the **generalized learnings** from the first cycle. The data of the *IaC Analyzer Decision Guide* had to be publicly accessible to developers and used by the application simultaneously. A straightforward solution is to store the data in YAML files or formats akin to **YAML in a GitHub repository**. Thereby, developers can make updates by creating pull requests. The access management of GitHub is used instead of having to self-develop such a mechanism using a database.

Furthermore, the **Go programming language** is suitable for smaller APIs, as shown by *ValidKube*, *ValidIaC*, and the *IaC Analyzers Decision Guide* developed in this work. This observation is further underlined by the large number of SICAs using Go. Of course, the application could have been developed with other programming languages, but researchers and practitioners may consider Go for similar projects. The same applies to the React frontend, which allowed fast development from the start.

Second Cycle

The **generalized learnings** from the second cycle are that when using GitHub as the version control platform, **GitHub Actions** should be considered for implementing a CI/CD pipeline. It is easy to use, well documented, and, most importantly, the ecosystem of ready-to-use actions in the marketplace is enormous.

Besides, the **conventional commit guidelines** (Conventional Commits, 2022) offer a convenient guideline for formatting commits. These guidelines play well with other GitHub Actions to automatically generate a new version number and create a GitHub release. Such a mechanism is beneficial because developers do not need to manage version numbers. Instead, they must only format their commits according to the guidelines. A consistent commit format helps other developers collaborate.

Last, creating version numbers according to **semantic versioning** (Preston-Werner, n.d.) helps consumers of an application decide whether they want to use a newer version. For instance, when deciding to update to a newer major version, users are aware that there may be breaking changes, and it is accepted practice to review the release notes.

Third Cycle

One **generalized learning** from the third cycle is that a small application serving backend and frontend via the same Go API in one **single Docker Image** decreases the complexity of the deployment and its costs. Furthermore, **DigitalOcean's App Platform** is suitable for deploying ready-to-use Docker Images and well-integrated with the DigitalOcean *Terraform* provider. At €5 per month, it is reasonably inexpensive to use DigitalOcean's App platform. Nevertheless, it is currently impossible to configure a DigitalOcean app resource to be part of a DigitalOcean project resource (i.e., namespace) with *Terraform*. Assigning resources to a project is only possible for other resource types like domains.

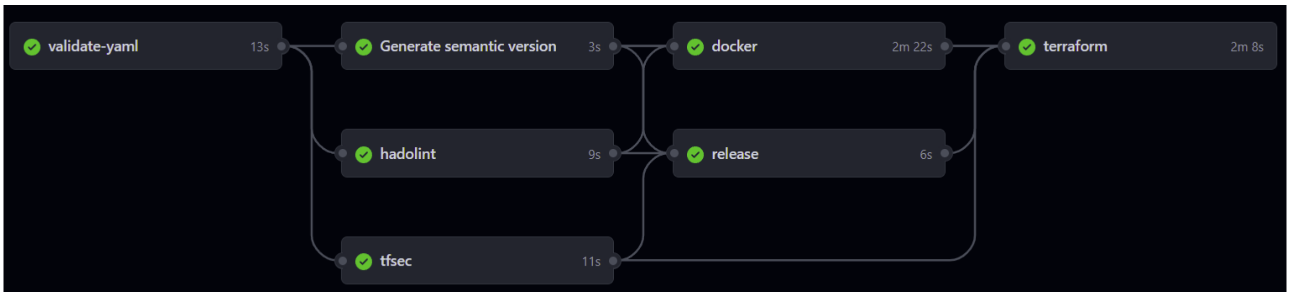
4.6.5. Result

The entire chapter was guided by the research question: **RQ 3.6**: *How can the collected data about static infrastructure code analyzers be used to develop an application which supports practitioners in finding suitable tools for their specific requirements?* It was answered by the development of the *IaC Analyzer Decision Guide*. The *IaC Analyzer Decision Guide* helps researchers and practitioners to find SICAs based on their requirements. Practitioners can identify SICAs for their IaC projects, whereas researchers can use the *IaC Analyzer Decision Guide* to find future work directions.

The application is available online at **https://iac-analyzers.dev**. The smallest possible CPU and memory sizes were chosen for the deployed Docker container (512 MB memory and one vCPU). When no requests are sent to the application, the CPU usage is at 0%, and the memory usage is at 14%, corresponding to about 72 MB of memory.

Although the developed application is small and has no users, the **CI/CD pipeline** was considered from the beginning. Due to that, the risk of introducing security-related issues in the *Dockerfile* or the *Terraform* manifests was reduced. Even more importantly, pushing code to the main branch deploys a new version of the application within less than ten minutes. How vital such fast deployment times are, has been discussed by Forsgren et al. (2018) in-depth. The final CI/CD pipeline is depicted in Figure 22.

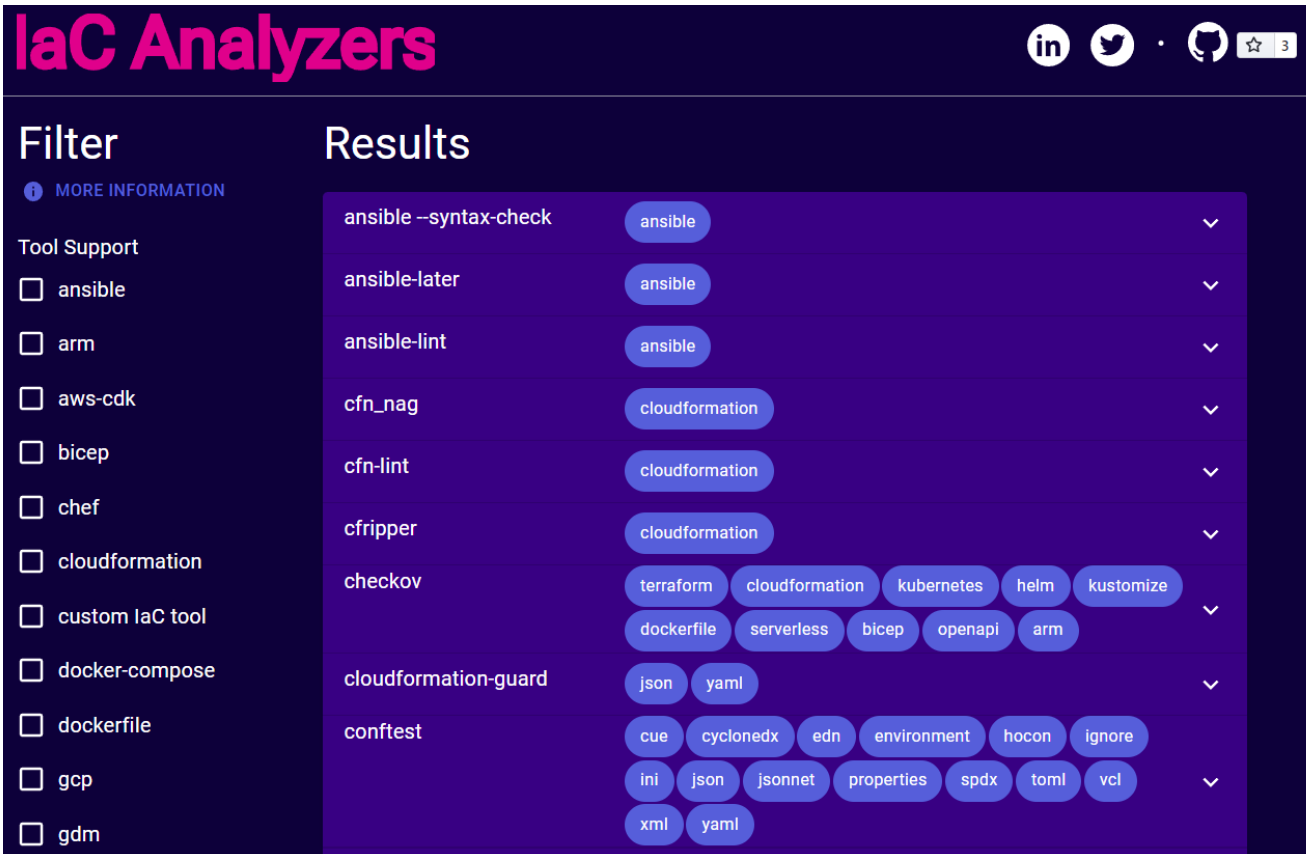
Figure 22. IaC Analyzer Decision Guide: CI/CD Pipeline



Source: Own representation.

Figure 23 shows a section of the user interface of the *IaC Analyzer Decision Guide*, where the users can select options matching their requirements. For instance, they can choose the IaC tool they use. Only two option categories are shown on this page by default ("Rules" is not visible in Figure 23), but others may be added in the future if users give that feedback. The categories "File Support" and "Rule Implementation" only appear when users select the "custom IaC tool" and "customChecks" options, respectively.

Figure 23. IaC Analyzer Decision Guide



IaC Analyzers

Filter

MORE INFORMATION

Tool Support

- ☐ ansible
- ☐ arm
- ☐ aws-cdk
- ☐ bicep
- ☐ chef
- ☐ cloudformation
- ☐ custom IaC tool
- ☐ docker-compose
- ☐ dockerfile
- ☐ gcp
- ☐ gdm

Results

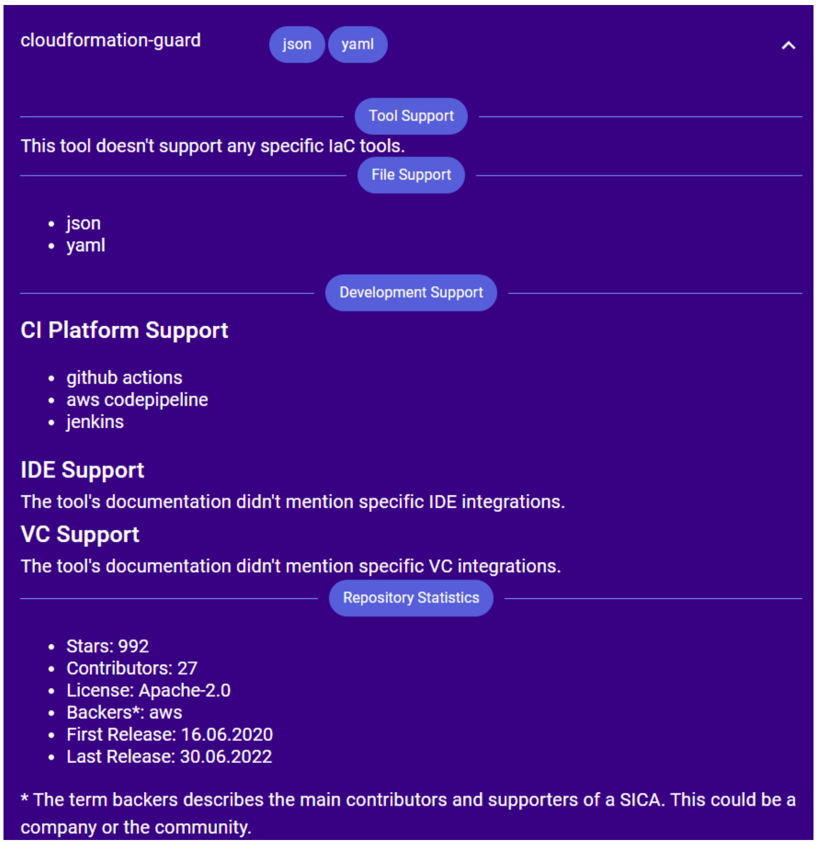
ansible --syntax-check	ansible
ansible-later	ansible
ansible-lint	ansible
cfn_nag	cloudformation
cfn-lint	cloudformation
cfripper	cloudformation
checkov	terraform, cloudformation, kubernetes, helm, kustomize, dockerfile, serverless, bicep, openapi, arm
cloudformation-guard	json, yaml
conftest	cue, cyclonedx, edn, environment, hocon, ignore, ini, json, jsonnet, properties, spdx, toml, vcl, xml, yaml

Source: Own representation.

The SICA data is displayed in so-called accordions (Material UI SAS, n.d.) because each SICA has a large section listing all the characteristics of the particular SICA. Displaying all this data by default would display too much information at once. The IaC tools and file types a particular SICA supports are displayed in the middle of each element. This is essential information, and users can decide which tools they want to investigate in detail based on it.

Clicking the arrow icon on the right side of a particular accordion opens the detail section. Figure 24 shows a section of the collapsed accordion element for the SICA *cloudformation-guard*. Evidently, displaying all that data for each SICA by default would likely overwhelm users.

Figure 24. IaC Analyzer Decision Guide: Details Section of cloudformation-guard


cloudformation-guard json yaml

Tool Support

This tool doesn't support any specific IaC tools.

File Support

- json
- yaml

Development Support

CI Platform Support

- github actions
- aws codepipeline
- jenkins

IDE Support

The tool's documentation didn't mention specific IDE integrations.

VC Support

The tool's documentation didn't mention specific VC integrations.

Repository Statistics

- Stars: 992
- Contributors: 27
- License: Apache-2.0
- Backers*: aws
- First Release: 16.06.2020
- Last Release: 30.06.2022

* The term backers describes the main contributors and supporters of a SICA. This could be a company or the community.

Source: Own representation.

To this point, it was described how users could utilize the graphical user interface to select options and how the *IaC Analyzer Decision Guide* returns matching SICAs. In addition, an **API endpoint** provides statistics about the SICAs contained in the *IaC Analyzer Decision Guide*. The endpoint is accessible at https://iac-analyzers.dev/api/sicas/stats. With this endpoint, the current state of the findings reported in chapters 4.5.1 and 4.5.3 can be retrieved at any time. For instance, the endpoint returns data about the rule languages the SICAs use and how they can be installed. All available endpoints, including the data model, can be investigated on the OpenAPI UI (https://iac-analyzers.dev/swagger/index.html#/). Note that *https* must be selected as the schema when calling an endpoint via the OpenAPI UI.

Beyond that, the practical adoption of the *IaC Analyzer Decision Guide* has already started. The creator of *MegaLinter* (Vuillamy & OX Security, 2022) contacted the author of this work. He explained that he would create a reference to the *IaC Analyzer Decision Guide* in the *MegaLinter's* documentation and contribute to the application by refining the given information on the *MegaLinter* (N. Vuillamy, personal communication, LinkedIn message of September 8, 2022).

4.6.6. Future Work on the IaC Analyzer Decision Guide

Many extensions of the *IaC Analyzer Decision Guide* were suggested during the application development and usability test sessions. These features are shortly explained because they may be picked up by future research studies, the author, or open-source contributors.

First, a **feedback button** must be added to each SICA displayed so that users can quickly send feedback on improvements without having to fork the repository, update the specific YAML file, create a pull request, or create an issue on GitHub. Such a feedback button would help to identify wrong information. The feedback would then be sent via email (or a similar process) to the author of this work.

At the time of writing, only the most popular SICAs were integrated into the *IaC Analyzer Decision Guide*, whereas unpopular tools such as the *SLIC* or *SLAC* were omitted. These **unpopular tools must also be added** to the *IaC Analyzer Decision Guide* in the future. Thereby, the application will offer a comprehensive list of all known SICAs. Adding these tools is as simple as creating one YAML file for each SICA and extracting the characteristics of a SICA from its documentation and code.

An interesting topic for future work is to **generate a natural text** based on the YAML files via NLP. Such a text summarizing the most important characteristics of a SICA and its differences from other tools could help practitioners present a certain SICA to their team. It could also help researchers to define requirements for a SICA they develop.

Another interesting topic for future work is to investigate how the characteristics of the SICAs could be **automatically extracted from the documentation** and the code of these tools. Such a mechanism could update the *IaC Analyzer Decision Guide* automatically. Moreover, the generalized findings of such an approach would likely benefit other research areas.

Currently, users must manually compare SICAs they find during the search process (e.g., which tool has more stars on GitHub). A comparison dialog could improve this manual comparison by displaying two or more **tools side-by-side** in a table, highlighting the SICA with the best performance, if applicable. Referring to the example of the number of GitHub stars, the cell with the highest number of GitHub stars could be highlighted, which would help the user quickly identify the most popular tool.

Unit, integration, and end-to-end tests for the frontend and backend applications must be created to prevent a broken application version from being deployed to production. Also, a test environment where a new version can be tested before deploying to production is a future task. However, this task is more suitable for the author of the present work or open-source contributors than for a research study.

5. Conclusion

The three main research questions were answered within the six consecutive phases of this work (chapters 4.1 to 4.6). The **first research question** about existing SICAs and their characteristics was answered by identifying the most popular SICAs via a multivocal literature review and analyzing the characteristics of the tools. The characteristics of the SICAs were extracted by studying the SICAs' documentation and code and executing them. Based on the findings, it can be concluded that many SICAs exist, in contrast to what was explained by researchers and practitioners. Furthermore, the more popular an IaC tool is, the more SICAs are created for it. Unexpectedly, over 40 popular SICAs were identified, and many are highly customizable. For instance, 27 tools support custom rules. The findings revealed that practitioners created the most popular SICAs. Those SICAs developed by researchers are hardly suitable for practical use. Mapping SICAs with IaC tools and categories (e.g., security) answered the **second research question** about gaps in static infrastructure code analysis. It revealed several understudied areas. The results indicate that although many SICAs exist, several IaC tools (e.g., *Nomad*) and categories (e.g., resource visualization for provisioning tools) are systematically understudied. In essence, coverage for the most popular IaC tools is better than for less popular ones. The understudied areas are elaborated on below. The **third research question** about how to publish the collected data was answered by developing an interactive web application. The *IaC Analyzer Decision Guide* allows practitioners to identify SICAs for their projects and gives researchers an overview of the current state of practice and research. It offers an easy way to avoid creating SICAs for already covered use cases. The *IaC Analyzer Decision Guide* can be easily updated and automatically deployed within less than ten minutes.

This work used **multivocal literature reviews** to identify IaC tools and SICAs and was the first in static infrastructure code analysis research that applied this methodology. It allowed the inclusion of formal and informal literature. Since the body of formal literature is small compared to the informal literature, MLRs proved to be a purposeful methodology. The collected data needed to be published in a format suitable for researchers and practitioners that allowed the data to be constantly updated. Hence, the *IaC Analyzer Decision Guide* was developed with the **design science research** methodology. Its iterative nature allowed constant improvement to the application. Nevertheless, reporting the results of the design science research project was challenging due to the lack of guidelines on an exact reporting format. To sum it up, the multivocal literature reviews and design science research approach directed an effective procedure to answer the research questions. The conducted **usability test sessions** proved to be an effective way to evaluate the usability of an application

Due to the limited time, only the **most popular IaC tools and corresponding SICAs** were analyzed. Hence, the results described above are not generalizable to all SICAs. Still, it is assumed that practitioners and researchers aim to develop popular SICAs so that they might orient their work towards the most popular SICAs, not the less popular ones. The **in-depth assessment of the SICAs is biased by the raters' judgment**. The author mitigated this risk using two raters and calculating Cohen's Kappa score. Similarly, the **extraction of characteristics from the SICAs' code and documentation was subject to the interpretation of the author** of this work. The author mitigated this risk by publishing the data on GitHub, which allows misinterpretations to be resolved by affected SICA developers.

This thesis has mentioned many research topics on improvements to the *IaC Analyzer Decision Guide*. These recommendations have already been explained in chapter 4.6.6. Beyond that, mapping IaC tools and categories to the identified SICAs in chapter 4.5.4 suggested several topics for future work. First, no **static analyzers for *Pulumi*** were identified. Chapter 4.5.4 already elaborated on this topic, and possible solutions were explained, which will not be repeated here. Second, even if a SICA for a particular tool and category exists, it does not mean the SICA covers all use cases. For instance, many *Terraform* SICAs have only rules for the most popular public cloud providers like *AWS*. This observation suggests **developing rules for *Terraform* providers of less popular cloud providers** like *DigitalOcean*. As shown in chapter 4.5.4, developing such custom rules, for instance, with *tfsec*, is often straightforward. The main effort is to identify misconfigurations by analyzing existing infrastructure code or interviewing practitioners. Third, another understudied category for SICAs is **resource visualization**. These SICAs analyze the infrastructure manifests and visualize the encoded resources, which only makes sense for provisioning and orchestration tools. Still, such tools could only be identified for *Terraform* and *Docker Compose*. Fourth, **cost prediction** is understudied. *Infracost* already makes cost predictions for *Terraform* code of the most popular public cloud providers. However, the same would be interesting for *Terraform* code of less popular public cloud providers or native IaC tools of the cloud providers, like *CloudFormation*. Fifth, the IaC tool ***Nomad* is understudied**. Since *Nomad* uses its own DSL, static checks would be very similar to other SICAs. Moreover, some of the existing checks of other SICAs could be transferred to a static analyzer for *Nomad*, for instance, missing CPU and memory limits. Sixth, deciding which SICA has the best built-in rules is challenging. The *goat* repositories of bridgecrew are a good starting point to compare SICAs for the same IaC tool. However, the repositories may yield outstanding results for *checkov*, developed by bridgecrew. Hence, future work could develop a **standardized testing approach for SICAs** that allows comparison of the different tools and quantification of the quality of a SICA over time by a benchmarking-like approach with a repository that contains known vulnerabilities. Last, an **in-depth analysis of the built-in rules** of the SICAs could provide beneficial results. For instance, *tfsec* and *trivy* both integrate rules from the *DefSec* repository (Aqua Security Software Inc., 2021). How do the built-in *Terraform* rules of *checkov* compare to those of *tfsec*? Can the rules easily be transferred between the two tools, although the tools implement their rules differently (Python and JSON/YAML, respectively)? Could the rules be automatically transferred? Are the rules currently synchronized manually?

The **main contribution** of this work is the methodological approach using multivocal literature reviews, which is novel to formal static infrastructure code analysis research. It allowed the inclusion of many SICAs that have been systematically ignored in formal literature until this point. This work identified over **40 popular SICAs and their characteristics for the most popular IaC tools** using 370 formal and informal sources. The collected data was published via the ***IaC Analyzer Decision Guide***. Practitioners can use the application to find suitable SICAs for their projects, whereas researchers can use it to identify research gaps and seek inspiration when they develop novel SICAs. Researchers may also **incorporate their research findings (e.g., new rules) into existing SICAs** instead of creating new ones from scratch. This approach was rarely used in formal literature so far. Since data about existing SICAs is more accessible now than before, it might become a valuable option. Practitioners and researchers can work together by building feature-rich SICAs based on formal research-backed knowledge of researchers (rules) and experience-based knowledge of practitioners (tools).

IaC avoids manual errors during deployments but is still vulnerable to misconfigurations in the infrastructure code (Chiari et al., 2022). Static infrastructure code analysis is fast, license fee-free (if open-source tools are used), stable, reliable, and easy to use (Brikman, 2022, chapter 9). SICAs identify misconfigurations early in the development process. Practitioners must only find a SICA that meets their requirements, configure it, and incorporate it into the development process. Ultimately, this work **bridged the gap between research and practice** by collecting data about the most popular SICAs and their characteristics. The *IaC Analyzer Decision Guide* is available at https://iac-analyzers.dev and the repository at https://github.com/nileger/iac-analyzers.

V. Bibliography

23andMe. (2014). *Yamale: A schema and validator for YAML* [Source code]. 23andMe. https://github.com/23andMe/Yamale

Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M. L., & Stransky, C. (2017). Comparing the Usability of Cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy (SP)* (pp. 154–171). IEEE. https://doi.org/10.1109/SP.2017.52

Akula, M. (2020). *kubernetes-goat: Kubernetes Goat is a "Vulnerable by Design" cluster environment to learn and practice Kubernetes security* [Source code]. https://github.com/madhuakula/kubernetes-goat

Alnafessah, A., Gias, A. U., Wang, R [Runan], Zhu, L., Casale, G., & Filieri, A. (2021). Quality-Aware DevOps Research: Where Do We Stand? *IEEE Access*, *9*, 44476–44489. https://doi.org/10.1109/ACCESS.2021.3064867

Amazon Web Services. (n.d.–a). *Amazon EKS: Managed Kubernetes Service*. Retrieved May 13, 2022, from https://aws.amazon.com/eks/

Amazon Web Services. (n.d.–b). *Amazon RDS: Fully Managed Relational Database*. Retrieved September 11, 2022, from https://aws.amazon.com/rds/

Amazon Web Services. (n.d.–c). *AWS Cloud Development Kit*. Retrieved September 2, 2022, from https://aws.amazon.com/cdk/

Amazon Web Services. (n.d.–d). *AWS CloudFormation* [Computer software]. https://aws.amazon.com/cloudformation/

Analysis Tools. (2021). *A curated list of static analysis (SAST) tools for all programming languages, config files, build tools, and more* [Source code]. Analysis Tools. https://github.com/analysis-tools-dev/static-analysis/

Andrzejewski, D., Breitschwerdt, R., Fellmann, M., & Beck, E. (2017). Supporting breast cancer decisions using formalized guidelines and experts decision patterns: Initial prototype and evaluation. *Health Information Science and Systems*, *5*(1), 12. https://doi.org/10.1007/s13755-017-0035-8

Aqua Security Software. (n.d.). *tfsec* [Computer software]. https://aquasecurity.github.io/tfsec/v1.27.6/

Aqua Security Software Inc. (2021). *defsec: DefSec is a set of tools for scanning definitions of infrastructure* [Source code]. Aqua Security Software Inc. https://github.com/aquasecurity/defsec

Armstrong, J. (n.d.). Infrastructure as Code in a DevSecOps World. *Snyk*. https://snyk.io/learn/infrastructure-as-code-iac/

Ashfaq, Q., Khan, R., & Farooq, S. (2019). A Comparative Analysis of Static Code Analysis Tools that check Java Code Adherence to Java Coding Standards. In *2019 2nd International Conference on Communication, Computing and Digital systems: 6-7 March 2019, Islamabad, Pakistan* (pp. 98–103). Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/C-CODE.2019.8681007

AWS Cloudformation. (2018). *cfn-lint: CloudFormation Linter* [Source code]. https://github.com/aws-cloudformation/cfn-lint

Benner-Wickner, M., Kneuper, R., & Schlömer, I. (2020). *Leitfaden für die Nutzung von Design Science Research in Abschlussarbeiten*. *IT & Engineering: Vol. 2*. IU Internationale Hochschule.

Bhuiyan, F. A., & Rahman, A. (2020). Characterizing co-located insecure coding patterns in infrastructure as code scripts. In J. Grundy, C. Le Goues, & D. Lo (Eds.), *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops* (pp. 27–32). ACM. https://doi.org/10.1145/3417113.3422154

Black, R. (2015). *Advanced software testing* (2nd ed.). Rocky Nook.

Blakeman, K. (2018). *Top Search Tips.* RBA information. http://www.rba.co.uk/search/TopSearchTips.pdf

Brassel, S., & Gadatsch, A. (2020). Einführung von Public Cloud Services – Herausforderungen und Lösungsansätze aus der Praxis. *HMD Praxis Der Wirtschaftsinformatik*, *57*(5), 949–960. https://doi.org/10.1365/s40702-020-00652-5

bridgecrew. (n.d.–a). *checkov* [Computer software]. https://www.checkov.io/

bridgecrew. (n.d.–b). *Quick Start*. Retrieved July 12, 2022, from https://www.checkov.io/1.Welcome/Quick%20Start.html

bridgecrew. (2020a). *cfngoat: "Vulnerable by Design" Cloudformation repository* [Source code]. bridgecrew. https://github.com/bridgecrewio/cfngoat

bridgecrew. (2020b). *State of Open Source Terraform Security Report: We analyzed the open source Terraform Registry to gauge its current security and compliance posture.* Prisma Cloud. https://bridgecrew.io/wp-content/uploads/state-of-open-source-terraform-security-2020.pdf

bridgecrew. (2020c). *terragoat: "Vulnerable by Design" Terraform repository* [Source code]. bridgecrew. https://github.com/bridgecrewio/terragoat

bridgecrew. (2021). *Suppressing and Skipping Policies* [Source code]. bridgecrew. https://github.com/bridgecrewio/checkov/blob/master/docs/2.Basics/Suppressing%20and%20Skipping%20Policies.md

bridgecrew. (2022). *S3PublicACLRead* [Source code]. bridgecrew. https://github.com/bridgecrewio/checkov/blob/master/checkov/terraform/checks/graph_checks/aws/S3PublicACLRead.yaml

Brikman, Y. (2022). *Terraform - Up and Running: Writing infrastructure as code*. O'Reilly Media.

Brown, W. J. (1998). *AntiPatterns: Refactoring software, architectures, and projects in crisis*. Wiley.

Canonical. (n.d.). *Multipass* [Computer software]. Retrieved September 3, 2022, from https://multipass.run/

Chiari, M., Pascalis, M. de, & Pradella, M. (2022). Static Analysis of Infrastructure as Code: a Survey. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)* (pp. 218–225). IEEE. https://doi.org/10.1109/ICSA-C54293.2022.00049

Cloud Native Computing Foundation. (n.d.–a). *Helm* [Computer software]. https://helm.sh/

Cloud Native Computing Foundation. (n.d.–b). *Kubernetes* [Computer software]. https://kubernetes.io/

Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, *20*(1), 37–46. https://doi.org/10.1177/001316446002000104

Coleman, M., Messier, A., Lochbaum, M., & Myers, R. (2014). *htmllint: Unofficial html5 linter and validator* [Source code]. https://github.com/htmllint/htmllint

Container Solutions. (2017). *kubernetes-examples: Minimal self-contained examples of standard* [Source code]. Container Solutions. https://github.com/ContainerSolutions/kubernetes-examples

Conventional Commits. (2022). *Conventional Commits*. https://www.conventionalcommits.org/en/v1.0.0/

Ćurković, M., & Košec, A. (2018). Bubble effect: Including internet search engines in systematic reviews introduces selection bias and impedes scientific reproducibility. *BMC Medical Research Methodology*, *18*(1), 130. https://doi.org/10.1186/s12874-018-0599-2

Dalla Palma, S., Di Nucci, D., Palomba, F., & Tamburri, D. A. (2021). Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics. *IEEE Transactions on Software Engineering*, 1. https://doi.org/10.1109/tse.2021.3051492

Dalla Palma, S., Di Nucci, D., & Tamburri, D. A. (2020). AnsibleMetrics: A Python library for measuring Infrastructure-as-Code blueprints in Ansible. *SoftwareX*, *12*, 100633. https://doi.org/10.1016/j.softx.2020.100633

DevSec Hardening Framework Team. (2018). *puppet-os-hardening: This puppet module provides numerous security-related configurations, providing all-round base protection* [Source code]. DevSec Hardening Framework Team. https://github.com/dev-sec/puppet-os-hardening

Docker. (n.d.). *Docker* [Computer software]. https://www.docker.com/

Dolstra, E., Vermaas, R., & Levy, S. (2013). Charon: Declarative provisioning and deployment. In *RELENG 2013: May 20, 2013, San Francisco, CA, USA : proceedings* (pp. 17–20). IEEE. https://doi.org/10.1109/RELENG.2013.6607691

Doolittle, J., & Blumen, R. (2022). Luke Hoban on Infrastructure as Code. *IEEE Software*, *39*(2), 112–114. https://doi.org/10.1109/MS.2021.3131919

Elmore, R. F. (1991). Comment on "Towards Rigor in Reviews of Multivocal Literatures: Applying the Exploratory Case Study Method". *Review of Educational Research*, *61*(3), 293-293-297. https://www.jstor.org/stable/1170632

Fábry, M. (2021). *Complete guide for picking the right tool for Terraform Security Code Analysis.* Revolgy. https://www.revolgy.com/insights/blog/complete-guide-for-picking-the-right-tool-for-terraform-security-code-analysis

Fatima, A., Bibi, S., & Hanif, R. (Jan. 2018). Comparative study on static code analysis tools for C/C++. In I. Staff (Ed.), *2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)* (pp. 465–469). IEEE. https://doi.org/10.1109/IBCAST.2018.8312265

Firefly. (2022). *validiac* [Source code]. Firefly. https://github.com/gofireflyio/validiac

Fontana, F. A., Dietrich, J., Walter, B., Yamashita, A., & Zanoni, M. (2016, March 14 - 2016, March 18). Antipattern and Code Smell False Positives: Preliminary Conceptualization and Classification. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (pp. 609–613). IEEE. https://doi.org/10.1109/SANER.2016.84

Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The science behind DevOps building and scaling high performing technology organizations*. IT Revolution.

Fowler, M. (1999). *Refactoring: Improving the design of existing code*. *The Addison-Wesley object technology series*. Addison-Wesley.

Fowler, M. (2012, July 10). *PhoenixServer*. https://martinfowler.com/bliki/PhoenixServer.html

Fowler, M. (2012, May 1). *Test Pyramid*. https://martinfowler.com/bliki/TestPyramid.html

Fryman, J. (2014). *puppet-nginx: Puppet Module to manage NGINX on various UNIXes* [Source code]. Vox Pupuli. https://github.com/voxpupuli/puppet-nginx

Garousi, V., Felderer, M., & Mäntylä, M. V. (2019). Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, *106*, 101–121. https://doi.org/10.1016/j.infsof.2018.09.006

Garousi, V., & Mäntylä, M. V. (2016). When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology*, *76*, 92–117. https://doi.org/10.1016/j.infsof.2016.04.015

Geary, C. (2020). *cloudblock* [Source code]. https://github.com/chadgeary/cloudblock

GitHub. (n.d.). *GitHub GraphQL API: GitHub Docs* (Free, Pro, & Team). https://docs.github.com/en/graphql

Group PSIH. (2020). *docker-compose-viz: Docker compose graph visualization* [Source code]. Group PSIH. https://github.com/pmsipilot/docker-compose-viz

Guerriero, M., Garriga, M., Tamburri, D. A., & Palomba, F. (2019). Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 580–589). IEEE. https://doi.org/10.1109/ICSME.2019.00092

hadolint. (2015). *hadolint: Dockerfile linter, validate inline bash, written in Haskell* [Source code]. https://github.com/hadolint/hadolint

Hall, T., Zhang, M [Min], Bowes, D., & Sun, Y. (2014). Some Code Smells Have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology*, *23*(4), 1–39. https://doi.org/10.1145/2629648

Halstenberg, J., Pfitzinger, B., & Jestädt, T. (2020). *DevOps*. Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-31405-7

Hasan, M. M., Bhuiyan, F. A., & Rahman, A. (2020). Testing practices for infrastructure as code. In N. Cardozo, I. Dusparic, M. Linares-Vásquez, K. Moran, & C. Escobar-Velásquez (Eds.), *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing* (pp. 7–12). ACM. https://doi.org/10.1145/3416504.3424334

HashiCorp. (n.d.–a). *Command: plan*. Retrieved August 4, 2022, from https://www.terraform.io/cli/commands/plan

HashiCorp. (n.d.–b). *Packer* [Computer software]. https://www.packer.io/

HashiCorp. (n.d.–c). *Terraform* [Computer software]. https://www.terraform.io/

Häußge, G. (n.d.). *A dev's guide to open source software licensing*. Retrieved August 4, 2022, from https://github.com/readme/guides/open-source-licensing

He, K. (2014). *dockerfiles: A collection of delicious docker recipes* [Source code]. https://github.com/vimagick/dockerfiles

Higgins, J. P. T., & Green, S. (2008). *Cochrane Handbook for Systematic Reviews of Interventions*. John Wiley & Sons. https://doi.org/10.1002/9780470712184

Hoang Thuan, N., Drechsler, A., & Antunes, P. (2019). Construction of Design Science Research Questions. *Communications of the Association for Information Systems*, 332–363. https://doi.org/10.17705/1CAIS.04420

Hortlund, A. (2021). *Security smells in open-source infrastructure as code scripts: A replication study* [Bachelor's thesis]. Karlstad University, Karlstadt, Sweden. https://www.diva-portal.org/smash/get/diva2:1564706/FULLTEXT01.pdf

Hummer, W., Rosenberg, F., Oliveira, F., & Eilam, T. (2013). Testing Idempotence for Infrastructure as Code. In D. Eyers & K. Schwan (Eds.), *LNCS sublibrary. SL 2, Programming and software engineering: Vol. 8275. Middleware 2012: ACM/IFIP/USENIX, 14th International Middleware Conference, Beijing, China, December 9-13, 2013 : proceedings / David Eyers, Karsten Schwan, (eds.)* (Vol. 8275, pp. 368–388). Springer. https://doi.org/10.1007/978-3-642-45065-5_19

Huyen, C. (2022). *Designing machine learning systems: An iterative process for production-ready applications*. O'Reilly.

Hyperjump. (2019). *grule-rule-engine* [Source code]. https://github.com/hyperjumptech/grule-rule-engine

iacsecurity. (2021). *tool-compare* [Source code]. iacsecurity. https://github.com/iacsecurity/tool-compare

Ibnelbachyr, K. (2021). *IaC static analysis tools for Terraform*. https://www.linkedin.com/pulse/iac-static-analysis-tools-terraform-khalid-ibnelbachyr/

Jamsa, K. (2022). *Cloud Computing* (2nd ed.). Jones & Bartlett Learning.

Kaur, A., & Nayyar, R. (2020). A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code. *Procedia Computer Science*, *171*, 2023–2029. https://doi.org/10.1016/j.procs.2020.04.217

Kitchenham, B., & Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering, *2*(EBSE 2007-001).

Knauss, E. (2021). Constructive Master's Thesis Work in Industry: Guidelines for Applying Design Science Research. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Joint Track on Software Engineering Education and Training: ICSE-JSEET 2021 : proceedings : virtual (originally Madrid, Spain), 25-28 May 2021* (pp. 110–121). IEEE Computer Society, Conference Publishing Services. https://doi.org/10.1109/ICSE-SEET52601.2021.00021

Koelsch, G. (2016). *Requirements writing for system engineering: Project success through realistic requirements*. Apress.

Koenig, A. (1998). Patterns and Antipatterns. In L. Rising (Ed.), *The Patterns Handbooks: Techniques, Strategies, and Applications* (pp. 383–389). Cambridge University Press.

Komodor. (2022). *validkube: ValidKube combines the best open-source tools to help ensure Kubernetes YAML best practices, hygiene & security* [Source code]. Komodor. https://github.com/komodorio/validkube

Kukharik, V. (2020). *postgresql_cluster: PostgreSQL High-Availability Cluster* [Source code]. https://github.com/vitabaks/postgresql_cluster

Kumara, I., Garriga, M., Romeu, A. U., Di Nucci, D., Palomba, F., Tamburri, D. A., & van den Heuvel, W.-J. (2021). The do's and don'ts of infrastructure code: A systematic gray literature review. *Information and Software Technology*, *137*, 106593. https://doi.org/10.1016/j.infsof.2021.106593

Laigner, R., Kalinowski, M., Mendença, D., & Garcia, A. (2021). *Updated Catalog of Java Dependency Injection Anti-Patterns.* https://doi.org/10.5281/ZENODO.4679322

Landis, J. R., & Koch, G. G. (1977). The measurement of observer agreement for categorical data. *Biometrics*, *33*(1), 159–174.

Lewis, W. (2017). *Software Testing and Continuous Quality Improvement* (3rd ed.). Auerbach Publications. https://doi.org/10.1201/9781439834367

Li, A., Yang, X., Kandula, S., & Zhang, M [Ming] (2011). Comparing Public-Cloud Providers. *IEEE Internet Computing*, *15*(2), 50–53. https://doi.org/10.1109/MIC.2011.36

Lisdorf, A. (2021). *Cloud Computing Basics: A Non-Technical Introduction*. Apress.

Luzar, A., Celozzi, G., & Cankar, M. (2021). *IaC Code security and components security inspection.* PIACERE.

Mäntylä, M. V., & Lassenius, C. (2007). Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, *11*(3), 395–431. https://doi.org/10.1007/s10664-006-9002-8

Material UI SAS. (n.d.). *React Accordion component: Material UI*. Retrieved August 3, 2022, from https://mui.com/material-ui/react-accordion/

Meijer, B., Hochstein, L., & Moser, R. (2022). *Ansible: Up and running* (3rd ed.). O'Reilly Media.

Molich, R., Riedemann, C., & Daske, L. (2015). *CPUX-UT Usability Test Report Example* (v0133). UXQB. https://uxqb.org/public/documents/CPUX-UT_EN_Usability-Test-Report-Example.pdf

Morris, K. (2013, June 13). *ImmutableServer*. https://martinfowler.com/bliki/ImmutableServer.html

Morris, K. (2020). *Infrastructure as code: Dynamic systems for the cloud age* (2nd ed.). O'Reilly.

Myrbakken, H., & Colomo-Palacios, R. (2017). DevSecOps: A Multivocal Literature Review. In A. Mas, A. Mesquida, R. V. O'Connor, T. Rout, & A. Dorling (Eds.), *Communications in Computer and Information Science. Software Process Improvement and Capability Determination* (Vol. 770, pp. 17–29). Springer International Publishing. https://doi.org/10.1007/978-3-319-67383-7_2

Nath, S. (2021, June 16). *Continuous Security for IaC in GitOps* [PowerPoint slides]. Software Engineering Institute. https://resources.sei.cmu.edu/asset_files/Presentation/2021_017_001_735186.pdf

Netlify. (n.d.). *Netlify* [Computer software]. https://www.netlify.com/

Niels, A. (2021). *Usability Test Templates* [Course material]. IU University of Applied Sciences. Project: Human Computer Interaction (DLMAIEUIUX02).

Novak, J., Krajnc, A., & Žontar, R. (2010). Taxonomy of static code analysis tools. In *The 33rd International Convention MIPRO*.

OpenShift. (2016). *openshift-ansible: Install and config an OpenShift 3.x cluster* [Source code]. OpenShift. https://github.com/openshift/openshift-ansible

Österle, H. (Ed.). (2010). *Gestaltungsorientierte Wirtschaftsinformatik: Ein Plädoyer für Rigor und Relevanz*. Infowerk.

Österle, H., Becker, J., Frank, U., Hess, T., Karagiannis, D., Krcmar, H., Loos, P., Mertens, P., Oberweis, A., & Sinz, E. J. (2011). Memorandum on design-oriented information systems research. *European Journal of Information Systems*, *20*(1), 7–10. https://doi.org/10.1057/ejis.2010.55

Özel, A., Pautz, T., & Schmidt, N. (2020). Infrastructure as Code als Maßnahme zur Cloud Automatisierung – Hilfestellung zur Auswahl des richtigen Werkzeugs. *HMD Praxis Der Wirtschaftsinformatik*, *57*(5), 936–948. https://doi.org/10.1365/s40702-020-00657-0

Parsick, S. (2020). *Automated Quality Assurance for Ansible Playbooks.* Cloudical Deutschland GmbH. https://the-report.cloud/automated-quality-assurance-for-ansible-playbooks

Pereira dos Reis, J., Brito e Abreu, F., Figueiredo Carneiro, G. de, & Anslow, C. (2022). Code Smells Detection and Visualization: A Systematic Literature Review. *Archives of Computational Methods in Engineering*, *29*(1), 47–94. https://doi.org/10.1007/s11831-021-09566-x

PR Newswire (2021, March 4). Bridgecrew shifts cloud security all the way left with real-time scanning and fixes in VS Code. *PR Newswire US*. https://www.prnewswire.com/news-releases/bridgecrew-shifts-cloud-security-all-the-way-left-with-real-time-scanning-and-fixes-in-vs-code-301240312.html

Preston-Werner, T. (n.d.). *Semantic Versioning 2.0.0*. Retrieved July 27, 2022, from https://semver.org/

Progress Software Corporation. (n.d.). *Chef Infra: Configuration Management System Software*. Retrieved May 14, 2022, from https://www.chef.io/products/chef-infra

Pulumi. (2022). *pulumi-cdk: Pulumi/CDK Interop Library* [Source code]. https://github.com/pulumi/pulumi-cdk/blob/main/examples/s3-object-lambda/lib/s3-object-lambda-stack.ts

Puppet. (n.d.). *Puppet* [Computer software]. https://puppet.com/

RabbitMQ. (2012). *chef-cookbook: Development repository for Chef cookbook RabbitMQ* [Source code]. RabbitMQ. https://github.com/rabbitmq/chef-cookbook

Rahman, A. (2018a). Anti-Patterns in Infrastructure as Code. In V. a. V. IEEE International Conference on Software Testing (Ed.), *2018 IEEE 11th International Conference on Software Testing, Verification and Validation: ICST 2018 : proceedings : 9-13 April 2018, Västerås, Sweden* (pp. 434–435). IEEE. https://doi.org/10.1109/ICST.2018.00057

Rahman, A. (2018b). Characteristics of defective infrastructure as code scripts in DevOps. In M. Chaudron, I. Crnkovic, M. Chechik, & M. Harman (Eds.), *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings* (pp. 476–479). ACM. https://doi.org/10.1145/3183440.3183452

Rahman, A. (2019). *SLIC-Ansible* [Source code]. https://github.com/rayhanur-rahman/SLIC-Ansible

Rahman, A., Barsha, F. L., & Morrison, P. (2021). Shhh! 12 Practices for Secret Management in Infrastructure as Code. In *2021 IEEE Secure Development Conference (SecDev)* (pp. 56–62). IEEE. https://doi.org/10.1109/SecDev51306.2021.00024

Rahman, A., Elder, S., Shezan, F. H., Frost, V., Stallings, J., & Williams, L. (2018, September 21). *Bugs in Infrastructure as Code*. https://arxiv.org/pdf/1809.07937

Rahman, A., Mahdavi-Hezaveh, R., & Williams, L. (2019). A systematic mapping study of infrastructure as code research. *Information and Software Technology*, *108*, 65–77. https://doi.org/10.1016/j.infsof.2018.12.004

Rahman, A., Parnin, C., & Williams, L. (2019). The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (pp. 164–175). IEEE. https://doi.org/10.1109/ICSE.2019.00033

Rahman, A., Partho, A., Morrison, P., & Williams, L. (2018). What questions do programmers ask about configuration as code? In J. Bosch, B. Fitzgerald, M. Goedicke, H. H. Olsson, M. Konersmann, & S. Krusche (Eds.), *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering* (pp. 16–22). ACM. https://doi.org/10.1145/3194760.3194769

Rahman, A., Rahman, M. R., Parnin, C., & Williams, L. (2019, July 16). *Security Smells in Ansible and Chef Scripts: A Replication Study*. https://arxiv.org/pdf/1907.07159

Rahman, M. R., Imtiaz, N., Storey, M.-A., & Williams, L. (2022). Why secret detection tools are not enough: It's not just about false positives - An industrial case study. *Empirical Software Engineering*, *27*(3), 59. https://doi.org/10.1007/s10664-021-10109-y

Rajapakse, R. N., Zahedi, M., Babar, M. A., & Shen, H. (2022). Challenges and solutions when adopting DevSecOps: A systematic review. *Information and Software Technology*, *141*, 106700. https://doi.org/10.1016/j.infsof.2021.106700

Rathee, S., & Chobe, A. (2022). Open Source Licenses. In S. Rathee & A. Chobe (Eds.), *Getting started with open source technologies: Applying open source technologies with projects and real use cases* (pp. 37–56). Apress. https://doi.org/10.1007/978-1-4842-8127-7_3

Red Hat. (n.d.). *Ansible* [Computer software]. https://www.ansible.com/

Regula. (2022, June 14). *Writing Rules*. https://regula.dev/development/writing-rules.html

Reynolds, Z. (2017). *Identifying and Documenting False Positive Patterns Generated by Static Code Analysis Tools* [Master's thesis]. Purdue University, Indianapolis, Indiana. https://scholarworks.iupui.edu/bitstream/handle/1805/13533/thesis-reynolds-final.pdf;jsessionid=9AF8BA36C17454624F3F55177DA4AFCC?sequence=1

Ribeiro, M. (2022). *Learning DevSecOps: Integrating continuous security across your organization*. O'Reilly Media.

Ribeiro, T. J. A. (2021). *An IDE Plug-in to detect security vulnerabilities in Infrastructure-as-Code Scripts* [Master's thesis]. University of Porto, Porto. https://repositorio-aberto.up.pt/bitstream/10216/135479/2/487161.pdf

Rushgrove, G. (2020, February 20). *Shifting Terraform Configuration Security Left* [Video]. Youtube. https://www.hashicorp.com/resources/shifting-terraform-configuration-security-left

Saavedra, N., & Ferreira, J. F. (2022). *GLITCH: an Intermediate-Representation-Based Security Analysis for Infrastructure as Code Scripts.* arXiv. https://doi.org/10.48550/arXiv.2205.14371

Salazar, H. (2021). *Utilizing Static Analysis Testing for Infrastructure as Code.* Perficient. https://blogs.perficient.com/2021/09/22/utilizing-static-analysis-testing-for-infrastructure-as-code/

Schermann, G., Zumberi, S., & Cito, J. (2018). Structured information on state and evolution of dockerfiles on github. In A. Zaidman, Y. Kamei, & E. Hill (Eds.), *Proceedings of the 15th International Conference on Mining Software Repositories* (pp. 26–29). ACM. https://doi.org/10.1145/3196398.3196456

Schoster, B. (2020). *Checkov: Security & Compliance for Your Infrastructure-as-Code* [Video]. Youtube. https://www.youtube.com/watch?v=n5EdM-e-9DU

Schwarz, J., Steffens, A., & Lichter, H. (2018). Code Smells in Infrastructure as Code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)* (pp. 220–228). IEEE. https://doi.org/10.1109/QUATIC.2018.00040

Scribd. (2022, March 8). *Current enterprise public cloud adoption worldwide from 2017 to 2022, by service* [Graph]. Statista. https://www.statista.com/

Sekhon, H. (2016). *Dockerfiles: 50+ DockerHub public images* [Source code]. https://github.com/HariSekhon/Dockerfiles

Seymour, D. (2020). *IAC Static Analysis with Bridgecrew.* Software Engineering Daily. https://softwareengineeringdaily.com/2020/08/20/iac-static-analysis-with-bridgecrew/

Shambaugh, R., Weiss, A., & Guha, A. (2015, September 17). *Rehearsal: A Configuration Verification Tool for Puppet.* arXiv. https://arxiv.org/pdf/1509.05100

Sharpe, T. (2022). *puppet-lint: Check that your Puppet manifests conform to the style guide* [Source code]. https://github.com/rodjek/puppet-lint

Shubin, J. (2015). *mgmt: Next generation distributed, event-driven, parallel config* [Source code]. https://github.com/purpleidea/mgmt

Siebra, C., Lacerda, R., Cerqueira, I., P. Quintino, J., Florentin, F., Da Q. B. Silva, F., & L. M. Santos, A. (2018). From Theory to Practice: The Challenges of a DevOps Infrastructure as Code Implementation. In *Proceedings of the 13th International Conference on Software Technologies* (pp. 427–436). SCITEPRESS - Science and Technology Publications. https://doi.org/10.5220/0006826104270436

Sous Chefs. (2012). *aws: Development repository for the aws cookbook* [Source code]. https://github.com/sous-chefs/aws

Stack Overflow. (2022). *Stack Overflow Developer Survey 2022*. https://survey.stackoverflow.co/2022/

Stefanovic, D., Nikolic, D., Dakic, D., Spasojevic, I., & Ristic, S. (2021). Static Code Analysis Tools: A Systematic Literature Review. In B. Katalinic (Ed.), *DAAAM Proceedings. Proceedings of the 32nd International DAAAM Symposium 2021* (Vol. 1, pp. 565–573). DAAAM International Vienna. https://doi.org/10.2507/31st.daaam.proceedings.078

Tafani-Dereeper, C. (2020). *Scanning Infrastructure as Code for Security Issues*. https://blog.christophetd.fr/shifting-cloud-security-left-scanning-infrastructure-as-code-for-security-issues/

Taptu. (n.d.). *Curated list of awesome lists*. Retrieved July 1, 2022, from https://project-awesome.org/TaptuIT/awesome-devsecops

Tecnalia. (n.d.). *Piacere: Public Deliverables*. Retrieved September 3, 2022, from https://www.piacere-project.eu/public-deliverables

terraform-compliance. (n.d.). *Tags*. Retrieved July 19, 2022, from https://terraform-compliance.com/pages/Examples/AWS/tags_related.html#ensure-all-resources-have-the-whitelist-tag

Thames, W. (2014). *ansible-lint: Best practices checker for Ansible* [Source code]. Ansible. https://github.com/ansible/ansible-lint

Thomas, C. G. (2021). *Research Methodology and Scientific Writing* (2nd ed.). Springer.

Travis CI. (n.d.). *Travis CI* [Computer software]. https://www.travis-ci.com/

Turbonomic. (2021, March 2). *Global use of cloud providers by organizations 2021, by vendor* [Chart]. Statista. https:/www.statista.com/

UXQB. (n.d.). *Documents*. Retrieved August 5, 2022, from https://uxqb.org/en/documents/

Vajjala, S., Majumder, B., Gupta, A., & Surana, H. (2020). *Practical natural language processing: A comprehensive guide to building real-world NLP systems*. O'Reilly.

van der Bent, E., Hage, J., Visser, J., & Gousios, G. (2018). How good is your puppet? An empirically defined and validated quality model for puppet. In E. a. R. IEEE International Conference on Software Analysis (Ed.), *25th IEEE International Conference on Software Analysis, Evolution and Reengineering: SANER 2018 : Campobasso, Italy : proceedings* (pp. 164–174). IEEE. https://doi.org/10.1109/SANER.2018.8330206

Vergé, A. (2016). *yamllint: A linter for YAML files* [Source code]. https://github.com/adrienverge/yamllint

Vuillamy, N., & OX Security. (2022). *MegaLinter* [Computer software]. https://oxsecurity.github.io/megalinter/latest/

Walikar, R. (2022, March 11). *Fixing the default insecure network connection option for RDS instances*. https://kloudle.com/academy/fixing-the-default-insecure-network-connection-option-for-rds-instances

Wang, R [Rosemary]. (2022). *Infrastructure as code, patterns and practices: With examples in Python and Terraform*. Manning Publications.

Warkhade, A. (2022). *Testing your Infrastructure as Code using Terratest.* InfraCloud Technologies. https://www.infracloud.io/blogs/testing-iac-terratest/

Warpnet B.V. (2020). *salt-lint: A command-line utility that checks for best practices in SaltStack* [Source code]. https://github.com/warpnet/salt-lint

widdix. (2015). *aws-cf-templates: Free Templates for AWS CloudFormation* [Source code]. widdix. https://github.com/widdix/aws-cf-templates

Wilson, G. (2020). *DevSecOps: A leader's guide to producing secure software without compromising flow, feedback and continuous improvement*. Rethink Press.

Winkler, S. (2021). *Terraform in action*. Manning Publications.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In M. Shepperd (Ed.), *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (pp. 1–10). ACM. https://doi.org/10.1145/2601248.2601268

Yamashita, A., & Moonen, L. (2013). To what extent can maintenance problems be predicted by code smell detection? – An empirical study. *Information and Software Technology*, *55*(12), 2223–2242. https://doi.org/10.1016/j.infsof.2013.08.002

Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., & Di Penta, M. (2017). How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (pp. 334–344). IEEE. https://doi.org/10.1109/MSR.2017.2

VI. Appendices

Appendix A
Informed Consent Declaration

The informed consent declaration was adapted from Niels (2021).

I hereby give my permission that my behavior via Microsoft Teams screen share and video will be analyzed as part of my voluntary participation in a usability study of the **IaC Analyzer Decision Guide** on **August 19th** on **Microsoft Teams**.

I understand that

- The session will last about 15 minutes.
- My name will not be reported in association with the usability study.
- I am not being evaluated, but the **IaC Analyzer Decision Guide** is being evaluated.

I understand that if I do not want to continue, I can leave at any time during this session. I can also deny the consent at any time during or after the session by informing the moderator.

Place and Date	. . . .
Name (print)	. . . .
Signature	. . . .

Appendix B

Usability Test Plan

Origin of the Template

The usability test plan uses the template of Niels (2021), which is in the style of the CPUX-UT certification (UXQB, n.d.). The template was adapted to the size of the *IaC Analyzer Decision Guide* and the role of the usability test sessions in this work. The template was copied, the placeholders were adapted, and missing information was inserted.

Goals

The overall aim of the usability test is to find usability problems with the *IaC Analyzer Decision Guide* so that they can be fixed after the third cycle.

The test has the following specific goals:

- Is the user interface complex enough to meet the needs of experienced developers?
- Does the application help researchers find research gaps in static infrastructure code analysis?
- Does the application contain major usability flaws that prevent users from finding SICAs?

Test Object

The application under test is

- *IaC Analyzer Decision Guide*
- 0.0.5
- SICA search process

The key business goals of the *IaC Analyzer Decision Guide* are as follows:

- Help practitioners find SICAs that meet their requirements
- Help researchers find research gaps in static infrastructure code analysis

The application is targeted at practitioners and researchers of all age groups. These users have at least some experience with IaC and professional interest in static infrastructure analysis tools. Users utilize the *IaC Analyzer Decision Guide* to achieve the following goals:

- List all SICAs
- Filter SICAs based on the selected requirements

The environment in which the *IaC Analyzer Decision Guide* is used is as follows:

- The application is opened on laptops or larger monitors.

This is the environment that is simulated in the usability test.

Participants and Responsibilities

Three test participants are expected in the usability test session. Additionally, two backup participants are scheduled.

Participants will be practitioners or researchers experienced with IaC. The key characteristics of these participants are as follows:

- They have already worked on IaC projects
- They have been in the situation or are likely to be in the situation where they must introduce static infrastructure analysis

The key responsibilities of the author of this work in the usability test sessions are to:

- Arrange recruitment of suitable participants
- Prepare test scenarios and tasks
- Ensure the application is available
- Stay unbiased throughout the whole testing session
- Find similarities and differences between participants' task processing
- Analyze the results of the usability test sessions
- Derive conclusions and recommendations from the analysis

Test Location and Time Plan

The usability test will be carried out online via an online collaboration tool.

The used online collaboration tool will be Microsoft Teams. Each participant will be sent an individual link and appointment to prevent interruption by participants joining too early.

Each test session will last approximately 15 minutes.

Appendix C
Usability Test Script

Origin of the Template

The usability test script uses the template of Niels (2021), which is in the style of the CPUX-UT certification (UXQB, n.d.). The template was adapted to the size of the *IaC Analyzer Decision Guide* and the role of the usability test sessions in this work. The template was copied, the placeholders were adapted, and missing information was inserted.

General information

The usability test sessions will not be recorded because recordings are neither possible with the author's Capgemini MS Teams account nor with the IU MS Teams account. Furthermore, the author is experienced enough to take notes during the sessions without the need for a recording.

Screener Questions

- Exclusion questions
  - Are you fluent in English? (No -> reject)
- Domain Questions
  - Have you written an IaC manifest before? (No -> reject)
  - Have you more than one year of experience in IaC? (No -> reject)

Briefing

Hi, welcome, and thank you for coming. We would really like to know what you think about the *IaC Analyzer Decision Guide* and what does and does not work for you.

The procedure we are going to follow today goes like this. I am going to show you the *IaC Analyzer Decision Guide,* and you will use it. After you have used it, I will ask you to complete some questionnaires that aim to measure your opinion about the application. Finally, we will wrap up. The session will not last any longer than 15 minutes.

Now I would like you to sign a form for me. The form is what is called a statement of informed consent. It is a standard document I give to everyone I interview. It sets out your rights as a person who is participating in this kind of research. As a participant in this research:

- You may refuse to participate at any time.
- You may take a break at any time.
- You may ask questions at any time.
- Your answers are kept confidential.

Please read over the form, and if you are happy with the content, please sign it. Let me know if you have any questions.
Any questions before we begin?
Ok, after a short interview, we will take a look at the *IaC Analyzer Decision Guide*, but let me give you some instructions about how to approach it.

When you are using the *IaC Analyzer Decision Guide*, we would like you to "think out loud." This means we want you to tell us what you are thinking about as you use it. For example, we would like you to say what it is you are trying to do, what you are looking for, and any decisions you are making. If you get stuck or feel confused, we would like to hear that too.

Be as honest as possible. If you think something is awful, please say so. Do not be shy: you will not hurt anyone's feelings. Since the *IaC Analyzer Decision Guide* is designed for people like you, we really want to know what you think and what does and does not work for you.

My role is to communicate what you say to the design and development team. I will not be able to provide help or answer any questions. This is because we want to create the most realistic situation possible. Even though I will not be able to answer your questions, please ask them nonetheless. It is crucial that I hear all your questions and comments. I do not want to bias you toward liking or disliking the *IaC Analyzer Decision Guide* — so do not be surprised if sometimes I do not say anything in response to your comments or if my response is something neutral like "That is good feedback."

The most important thing to remember when you are using the *IaC Analyzer Decision Guide* is that you are testing the *IaC Analyzer Decision Guide* — the *IaC Analyzer Decision Guide* is not testing you. There is absolutely nothing that you can do wrong.

Pre-session interview questions

Ok, to begin with, I would like to find out a bit more about you:

- On average, how many hours per week do you work on IaC projects?
- What IaC tools do you know?
- Have you ever used any IaC testing approach?

Test Tasks

Now, I would like you to try a couple of things with the *IaC Analyzer Decision Guide*. Work just as you would typically, narrating your thoughts as you go along. Here is the list of things I would like you to do.

<u>Researchers</u>: Scenario: Imagine you are a researcher in the area of IaC. You are searching for a new research topic and found the *IaC Analyzer Decision Guide*. With the help of the application, you want to find a research gap that you can work on in your next research study.

<u>Practitioners:</u> Scenario: Imagine you are a senior software engineer working on an IaC project. Unfortunately, your infrastructure code showed some bugs in the past, which made their way to production. You want to prevent such misconfigurations in the future. Hence, you want to start testing your IaC code. Because IaC testing is complex, you decide to start with static analysis. Now, it is on

you to find a suitable tool. A friend tells you about the *IaC Analyzer Decision Guide*, which you now use to find the static analyzer you are going to integrate into the CI/CD pipeline in your project.

<u>All:</u> First, you want to see all available static analyzers to get a rough idea about the size of the ecosystem. Now, please use the *IaC Analyzer Decision Guide* to list all static analyzers.

<u>Practitioners:</u> In your project, you use *Terraform* and a *custom IaC tool* that is configured in JSON. Regarding the *Terraform* code, you want a static analyzer that has built-in rules. You are also sure that you will need to write your own rules because you have strict company regulations. The static analyzer for the custom IaC tool you use in your project must support custom rules too. Other than that, there are no requirements. Use the *IaC Analyzer Decision Guide* to find one or more static analyzers that cover all your requirements.

<u>Researchers:</u> You want to investigate how the existing static analyzers for *Ansible* are implemented. The programming language they are written in and the rule language they use to support custom rules are both of interest to you. Next, you are going to find out the same details about a few *Kubernetes* analyzers. Your research topic is then to develop a static analyzer for *Ansible* that is implemented in a similar way as most *Kubernetes* analyzers are, given analyzers for the two IaC tools are usually implemented differently. Which programming language and rule language would you use?

A task was considered completed when

- test participants answered the question from the test task

Any difficulties faced by the participant and associated verbatim comments will be noted on a data logging sheet. Each observation will comprise an observational code and a short description of the behavior.
Observational codes

- G - General comment
- P - Positive opinion
- N - Negative opinion
- B - Bug
- X - Usability problem

Debriefing questions
Considering the *IaC Analyzer Decision Guide* you have just used, please answer the following questions:

- Which two things are most in need of improvement?
- Which two things did you like most about the *IaC Analyzer Decision Guide*?
- Can you rate your experience on a five-point scale, where one is very dissatisfied and five is very satisfied?
- Can you tell me why you gave the task that rating?

Preparation checklist

- 1-3 weeks before the usability test
    - Choose a virtual collaboration tool
    - Write test plan
    - Write test script
    - Create screener interview
- One day before the usability test
    - Create one PDF with the task description of each of the tasks. Each task must be on one page.
    - Print test script.
    - Prepare consent form (send via email)
- Before each participant arrives
    - Send all required documents (consent form and tasks) and links (*IaC Analyzer Decision Guide*)
- Before starting the usability test session
    - Ask the participant to disable any software that might interrupt the test

Appendix D
Usability Test Report

Origin of the Template

The usability test report uses the template of Niels (2021), which is in the style of the CPUX-UT certification (UXQB, n.d.) and the sample usability test report (Molich et al., 2015). The template was adapted to the size of the *IaC Analyzer Decision Guide* and the role of the usability test sessions in this work. The template was copied, the placeholders were adapted, and missing information was inserted.

General Information

The usability test script and similar information that has already been explained in Appendix B and Appendix C are not repeated in this usability test report.

Executive Summary

The running version of the *IaC Analyzer Decision Guide* was usability tested in August 2022 with three target group members. The test method was the think-aloud method, where typical users carry out tasks while being observed.
The primary purpose of the usability test was to check whether the *IaC Analyzer Decision Guide* contains major usability flaws that prevent the completion of the SICA identification process. A secondary purpose was to check whether the user interface is complex enough to meet the requirements of experienced developers and whether the application helps researchers to find research gaps in static infrastructure code analysis.
This usability test report describes the findings and recommendations from the test.

Main positive findings for the *IaC Analyzer Decision Guide*:

- Participants consider the displayed information about the SICAs helpful in deciding on a tool.
- Participants like the general idea of the tool and find that it fulfills its purpose.
- Participants like the modern design of the application.

Main improvement areas for the *IaC Analyzer Decision Guide*:

- The filtering process is unintuitive
  Participants were confused about how to display all available SICAs. Furthermore, it was unclear which logical operator (“and” or “or”) was applied when selecting multiple options.
- The detailed data is displayed incomprehensively
  Participants found it challenging to identify information in the detail sections of the SICAs.
- The application expects too much knowledge from the users
  Some options are unclear to the users. They need descriptions of the options (e.g., ignore findings) and characteristics of the SICAs (e.g., backers). Furthermore, they need a short explanation of how to use the *IaC Analyzer Decision Guide*.

Legend

In the descriptions of findings, the following ratings are used:

Table D-1. Usability Test Session: Legend

✓	**Positive finding**	Works well. This approach is recommendable.
⊜	**Minor problem**	Minor dissatisfaction, noticeable delays, or superficial difficulties.
❗	**Major problem**	Substantial delays; or moderate dissatisfaction.
❌	**Critical problem**	Test participants gave up – showstopper; substantial dissatisfaction; or minor financial damage to the user.
💡	**Good idea**	A suggestion from a test participant or from the moderator that could lead to a significant improvement in the user experience.
⚡	**Functional problem**	Bug. The product does not work as specified.

Source: Niels (2021)

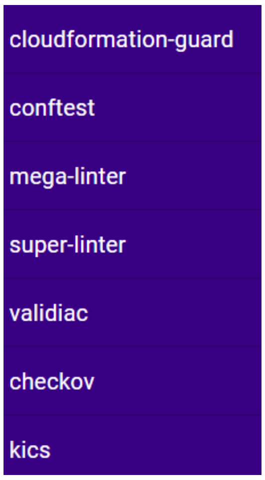
Conventions used in this usability test report

This usability test report contains several verbatim quotes from test participants. These quotes are formatted as follows:

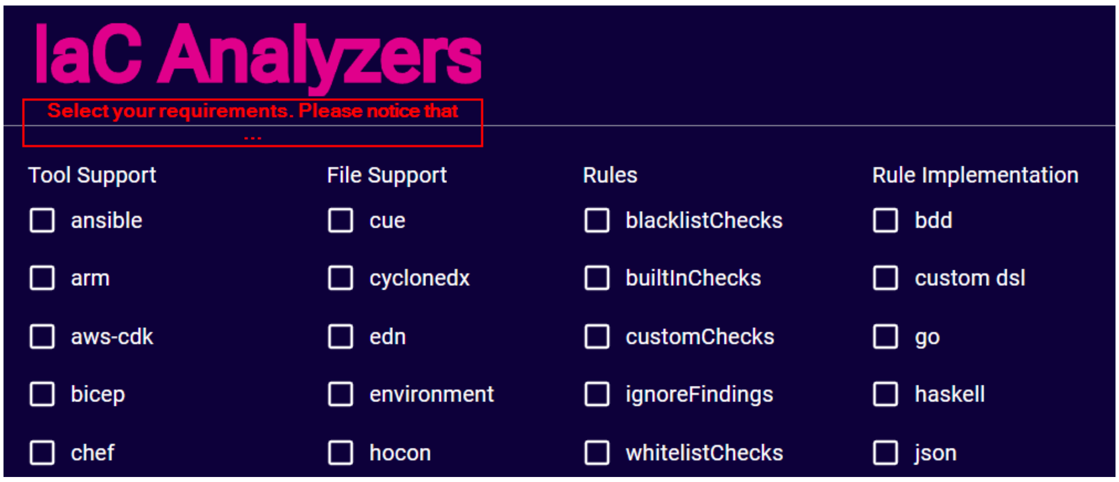
"Quotations appear in a separate line in quotation marks and italics."

Findings

Table D-2. Usability Test Session: Findings

⚡	**SICAs are not sorted alphabetically**
	Participants expected the SICAs to be sorted alphabetically.
	“I would expect the tools to be sorted alphabetically by default.”
	Sort the SICAs alphabetically.
	Figure D-1. Usability Test Session Finding: Sorting  cloudformation-guard conftest mega-linter super-linter validiac checkov kics Source: Own representation.

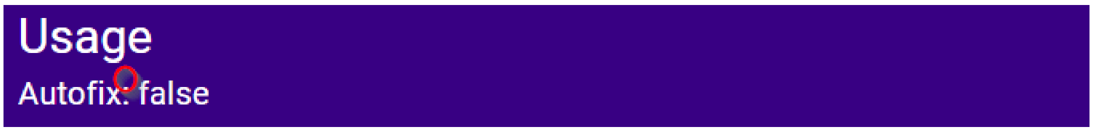
⚡	**The rule implementation filter does not work**
	Participants found that the rule implementation filter does not work. The output remains the same irrespective of what they choose.
	“It seems like the rule implementation options do not have any effect.”
	Repair the rule implementation filter, so that only SICAs that implement their rules in the selected languages are displayed.
❗	**Unclear how to show all existing SICAs**
	It was unclear to participants how to display all existing SICAs. The first thought they had was to select all options. After a while, they found that it is sufficient to select no option and click the search button.
	“It makes no sense that I have to select nothing but still get all SICAs.” *“Do I have to select all options now? I need an explanation.”* *“Why does the application display SICAs although I have not selected any option?”*
	Option 1: One option is to display all SICAs by default when opening the application. Option 2: Another option is to implement a select/deselect all button.
❗	**Participants are confused about what happens if they select multiple options**
	Participants were confused about how the search process works if they select multiple options. They asked whether the application searches for tools that have all the selected characteristics or only one of them.
	“Which logical operator is applied – ‘and’ or ‘or’?” *“How are the options connected if I select several options – ‘and’ or ‘or’?* *“I need an explanation of how the selection works.”*
	Display an explanation text above the filter criteria area.
	Figure D-2. Usability Test Session Finding: Info Text



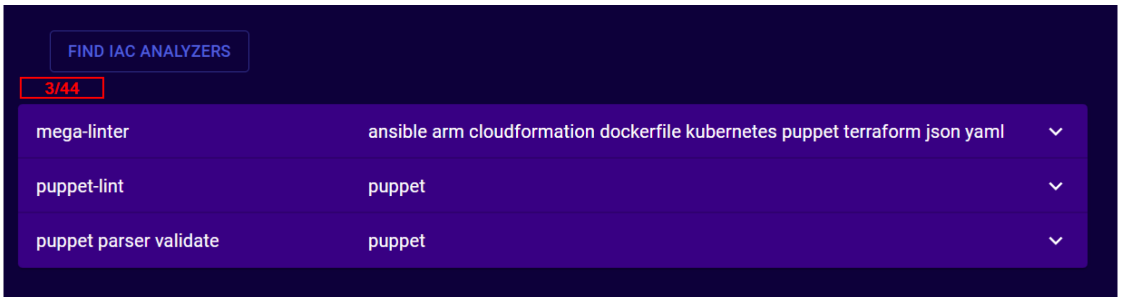
Tool Support	File Support	Rules	Rule Implementation
☐ ansible	☐ cue	☐ blacklistChecks	☐ bdd
☐ arm	☐ cyclonedx	☐ builtInChecks	☐ custom dsl
☐ aws-cdk	☐ edn	☐ customChecks	☐ go
☐ bicep	☐ environment	☐ ignoreFindings	☐ haskell
☐ chef	☐ hocon	☐ whitelistChecks	☐ json

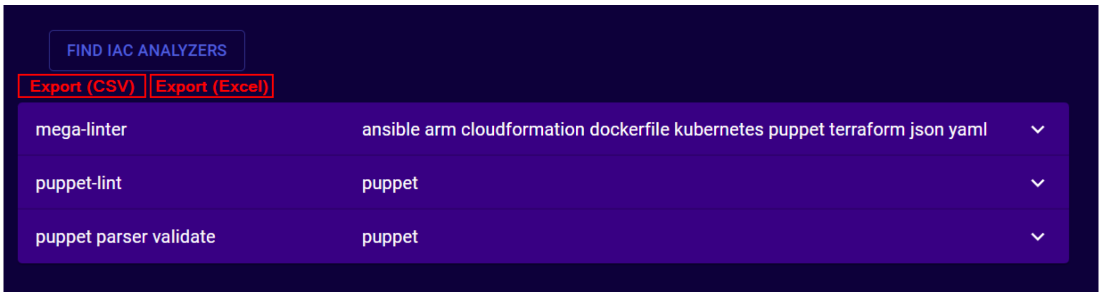
Source: Own representation.

❗	**Info texts for terms that could be unclear**
	Participants were confused by some terms they did not know. This applies to terms in the options and the detailed SICA information sections.

	“What does ‘autofix’ mean?” *“What does the term ‘backers’ mean?”* *“I am confused by the options in the category rules. What does ‘ignore findings’ mean, for instance?”*
	Option 1: Show small info icons next to terms that could be unclear. When users click on the info icon, they are given a short description of the term. Option 2: A legend at the bottom could also be sufficient and allow users to see the descriptions without clicking additional buttons.
	Figure D-3. Usability Test Session Finding: Info Texts for Unknown Terms Usage Autofix: false Source: Own representation.
❗	**Users find it challenging to locate data in the detailed information section per SICA**
	When users searched for information about the programming language and rule implementation language, they found it challenging to identify the information.
	“I find it complex to identify the information I need for the task.” *“I do not understand the pattern of the information section.”* *“I do not understand the different levels of headings. They are too similar.”* *“All information is too far on the left.”* *“I would expect the link to the GitHub repository below the GitHub statistics.”* *“How up-to-date are the GitHub statistics? I would expect an information text explaining that to me.”* *“I do not like how the data is displayed.”* *“The rule sections show too much information.”* *“I would like to see checkboxes instead of the words ‘true’ or ‘false.’”*
	Improve the visualization of the detailed information sections, for instance, by showing checkboxes instead of true and false.
❗	**Users are confused why the detailed information section of some SICAs missed some headings**
	Participants were confused why the detailed information section of some SICAs missed some headings. They expected to see the same layout for all SICAs.
	“I would expect that the headings are the same in all detail sections. And if there were no information to be displayed, I would expect a short explanation of why there is no information displayed.”
	A solution to this problem is to display the same structure for all SICAs. If there is no data for a particular heading, a short explanation must be provided to inform the user why no data is displayed.

	Example: File Support This tool only supports specific IaC tools and is unsuitable for scanning files of a particular type. If you search for such a tool, adapt your selection: Deselect all IaC tools and select the file type you are interested in (e.g., JSON).
🔵	**Users have to scroll to find the search button**
	Since there are many options users can select, participants searched for the search button because they had to scroll.
	“Where is the search button?” *“Okay, that search button is hard to find.”*
	Option 1: Display an overlay button in the bottom left corner, which is shown all the time. Option 2: Another option would be to omit the search button and automatically trigger a new search when users change the filter options.
🔵	**Do not show “rule implementation” options when the option “custom checks” is not selected**
	It makes no sense to show the “rule implementation” options by default. They are only relevant when users select “custom checks.”
	“Maybe the ‘rule implementation’ options should only be visible when I select ‘custom checks.’”
	Only show the “rule implementation” options when users select the “custom checks” option.
🔵	**Improve resetting the search**
	Participants wanted an option to remove all checks at once instead of clicking each check.
	“How can I get rid of all checks?”
	Add a button that allows deselecting all options (e.g., above the filter options).
🔵	**Better handling of custom IaC tools**
	Participants found it confusing that there is no option for a custom IaC tool. It was unclear to them that they had to select only the file type and no IaC tool.
	“Why is there no option ‘custom IaC tool’?
	Add the option “custom IaC tool” to the list of IaC tools or add an explanation of how to proceed if a custom IaC tool is used. Only show the file support options when users select the option “custom IaC tool.”
💡	**Display the number of SICAs for the current selection and the number of all available SICAs**
	Currently, users are given a list of SICAs that match their criteria. However, they do neither know how many SICAs are displayed (without manually counting) nor how many SICAs exist.

	“I would like to see how many SICAs are currently displayed versus how many exist in total.”
	Above the list of SICAs, a small text box could indicate how many of the total number of SICAs are displayed.
	Figure D-4. Usability Test Session Finding: Number of Identified SICAs  FIND IAC ANALYZERS 3/44 mega-linter — ansible arm cloudformation dockerfile kubernetes puppet terraform json yaml puppet-lint — puppet puppet parser validate — puppet Source: Own representation.
💡	**Display SICAs in tabular form**
	Currently, the SICAs are displayed below each other. Users have to collapse the accordions to show the detailed information section. To compare SICAs more efficiently, the SICAs could be displayed in a tabular form.
	“I would like to display the SICAs in a tabular form.”
	A toggle button above the list of SICAs could switch between different visualization options. Alternatively, the current visualization could be replaced by a tabular approach.
💡	**Comparison screen**
	Participants said they wanted to compare SICAs in a tabular comparison screen
	“I would like a table that compares those SICAs the IaC Analyzer Decision Guide displays.” *“A comparison option would be really helpful.”*
	Users could be given the option to select specific tools displayed on a tabular comparison screen.
💡	**Statistics about the displayed SICAs**
	Participants mentioned they wanted to see statistics about the SICAs the *IaC Analyzer Decision Guide* displays. For instance, when they search for *Terraform* SICAs, they want statistics about how these SICAs are implemented and what the most popular SICA is.
	“I would like to see statistics about these displayed tools.” *“To find out which programming language is the most popular among the currently displayed results, I would have to create a tally sheet. That is annoying.”*
	A statistics section could be displayed above the list of SICAs.
💡	**Export option**
	Participants wanted an option to export the SICAs (all SICAs or their specific selection) in another format with all details.

	They wanted to use the export to perform further analysis. Moreover, users would have a way to save their search results.
	“As a researcher, I would like to export the SICAs in CSV or Excel format because I need to perform further analysis. For instance, I would like to create pivot tables as a researcher. This option would be the honeypot.”
	Export buttons above the list of SICAs could allow exporting the search results into various formats.
	Figure D-5. Usability Test Session Finding: Export Option  mega-linter ansible arm cloudformation dockerfile kubernetes puppet terraform json yaml puppet-lint puppet puppet parser validate puppet Source: Own representation.
💡	**Links to IaC tools and other sources that need further explanation**
	Participants did not know what *OPA* (Open Policy Agent) was and came up with the idea to implement links for all mentioned terms that need further explanation
	“I would like a link here that directs me to an article explaining what OPA is.”
	Technically, this feature would be complex to implement. It could be implemented by creating a list of terms and corresponding links, which would be used to replace the text with a link.
💡	**Automatically change the output when users select options**
	Participants expected the output to change automatically when they selected different options.
	“I want the output to change automatically without clicking the search button.”
	Whenever the selected options change, automatically refresh the output.
💡	**Advanced sorting**
	Participants wanted to sort the results, for instance, by the number of GitHub stars. Thereby, participants wanted to select “the best” SICA in a specific category, for instance, popularity.
	“I would like to sort the results by the number of GitHub stars.” *“Maybe a button to show only the most popular tools would be helpful. Nevertheless, I do not want to define what exactly popular is. I would like to rely on the application.”* *“I need an option to sort the results by popularity, last commit date, and last release date.”*
	Change the output visualization to a tabular form where users can sort by columns, for example, the number of GitHub stars. Alternatively, add a dropdown menu where users can choose the characteristic used to sort the results.

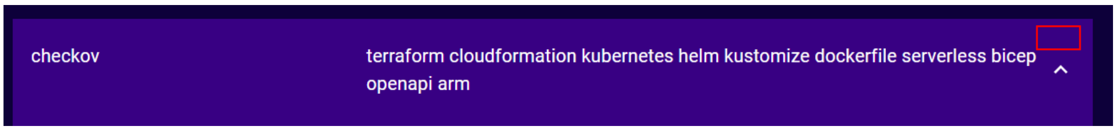
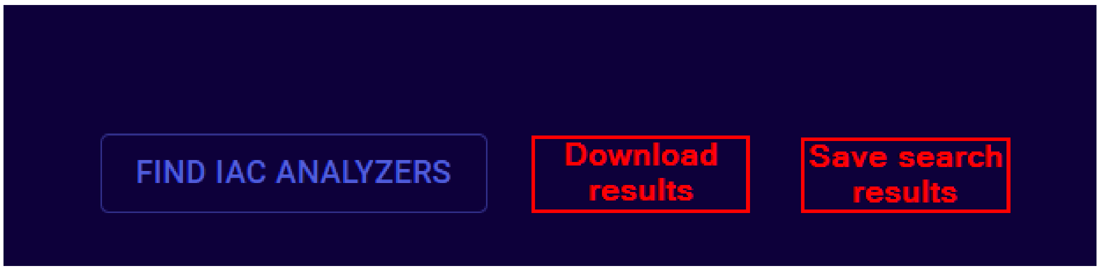
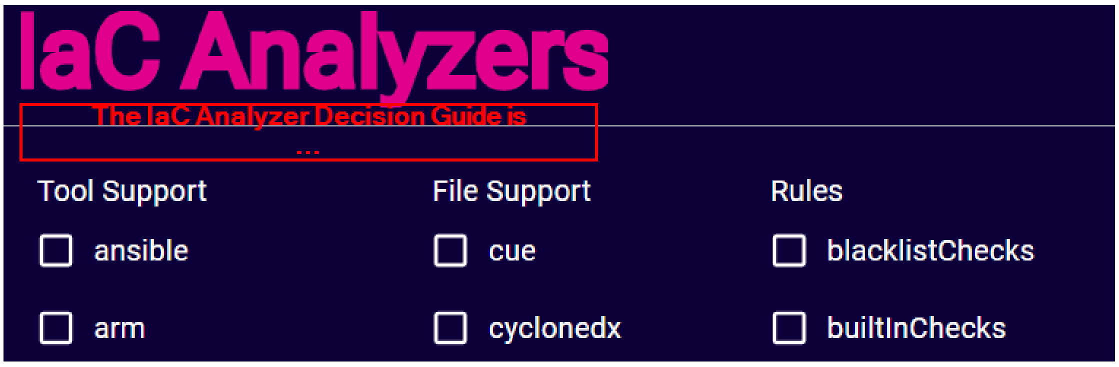
💡	**Feedback button**
	Participants wanted to have a feedback button per SICA to allow them to jump directly to the particular YAML file in the GitHub repository. They said that this would allow them to update SICA data faster.
	“I would like to have a feedback button for the SICAs.”
	Add a “Found a bug?” button to each SICA. This button could have an “edit” icon.
	Figure D-6. Usability Test Session Finding: Feedback Button  checkov terraform cloudformation kubernetes helm kustomize dockerfile serverless bicep openapi arm Source: Own representation.
💡	**Users need the option to save search results**
	Participants need an option to save their search results
	“Now, I must open a second tab to enter my requirements for another tool.” *“I would like to send my search results to others. Maybe the search criteria could be put into the URL when I search.”*
	Allow users to download their search results or save them in the application. Download and save buttons could be added next to the search button. When a user decides to store the search results, the results could be saved in the local store of the user’s browser. The search criteria could also be stored in the URL so that users can send their search results to others.
	Figure D-7. Usability Test Session Finding: Save Search Results  FIND IAC ANALYZERS Download results Save search results Source: Own representation.
💡	**Allow a list of additional resources**
	Participants wanted a list of additional resources.
	“I would like to have links to blog posts that explain how to use the SICA and how to integrate it into the development process.”
	Allow a list of blog posts and other resources to be added to each SICA.
💡	**A general explanation of how to use the application and what it is**
	Participants expected a short explanation of the application and how to use it.
	“I would like to have a description of what the IaC Analyzer Decision Guide is.”
	Display an explanation text above the filter criteria.

	Figure D-8. Usability Test Session Finding: How-To Use the IaC Analyzer Decision Guide  IaC Analyzers The IaC Analyzer Decision Guide is ... Tool Support: ☐ ansible ☐ arm File Support: ☐ cue ☐ cyclonedx Rules: ☐ blacklistChecks ☐ builtInChecks Source: Own representation.
💡	**Guided selection process**
	Participants found it challenging to identify the correct options. They suggested implementing a guided selection process feature.
	“The amount of options is overwhelming for me. I could imagine a guided process.”
	A stepper could guide the user through the process of selecting suitable SICAs.
💡	**Show more information when accordions are collapsed**
	In the accordion header, show more information.
	“I would like to see more information here. For instance, the number of GitHub stars would be interesting.”
	Add additional information to the accordion headers.
💡	**Add additional filter options**
	Participants asked why they had no additional filter options except the ones displayed. Still, they were not sure if they really needed more.
	“I would also like to filter by license.”
	Add additional filter options. However, the UI must not be overloaded. Hence, a suitable screen layout must be designed and tested first.
💡	**Dark and light theme**
	Participants said they would like to choose between a dark and light theme.
	“Many websites have a toggle to switch between dark and light theme.”
	Add a toggle in the top right corner to switch between the dark and light theme.
✓	**The detail section shows information that is interesting to users**
	Participants liked the information shown in the detail sections. They found it helpful for selecting a SICA.
	“Interesting, you always learn something new.”

Source: Own representation (changed) based on Niels, 2021.

Description of Object of Evaluation

The object of evaluation is the *IaC Analyzer Decision Guide* in version 0.0.5, as it was available to the public in August 2022. The *IaC Analyzer Decision Guide* is used by practitioners and researchers working on IaC projects and research studies, respectively.

The target group for the *IaC Analyzer Decision Guide* is practitioners and researchers working on IaC projects. These users are likely to be in the situation to decide on a particular static IaC analyzer or to identify a gap in static IaC analysis.

Test participants were asked to test

- the filtering process
- the detailed information section of the static infrastructure code analyzers

Thereby, all parts of the application were tested except the API calls (i.e., statistics).

Purpose of the Evaluation

The purpose of the evaluation was to identify usability problems in connection with the following general use cases and to highlight the strengths and weaknesses of the *IaC Analyzer Decision Guide*:

1. Select requirements from the available options
2. Filter static analyzers
3. Interpret the results

Evaluation Method

This usability test was conducted as moderated think-aloud usability test by Nils Leger, where typical users carry out tasks while being observed.

Three participants carried out two tasks with the *IaC Analyzer Decision Guide* in separate test sessions. At the end of each usability test session, they answered several pre-defined questions.

The tasks took between three to five minutes to complete. The total test time, including answering the questions after each test session, is approximately 15 minutes.

The notes were analyzed by Nils Leger.

The target group for the *IaC Analyzer Decision Guide* is practitioners and researchers working on IaC projects. These users are likely to be in the situation to decide on a particular static IaC analyzer or to identify a gap in static IaC analysis.

Usability Test Script

See Appendix C.

Findings for Each Participant

Table D-3. Findings for Each Participant: Legend

Legend	
✅	The task was solved correctly without problems.
⊜	Problems occurred which delayed the test participant in carrying out the task.
❗	The test participant encountered considerable problems but eventually succeeded in completing the task correctly.
❌	The test participant was unable to complete the task or arrived at a result that deviated significantly from the correct result.

Source: Niels (2021)

Table D-4. Findings for Each Participant: Summary

	Researcher	IaC developer	IaC expert
Task 1 Display all static analyzers	⊜	⊜	⊜
Task 2 Filter static analyzers	❗	⊜	⊜

Source: Own representation (changed) based on Niels, 2021.

Problems in task 1:

- Unclear how to show all existing SICAs
- Users have to scroll to find the search button

Problems in task 2:

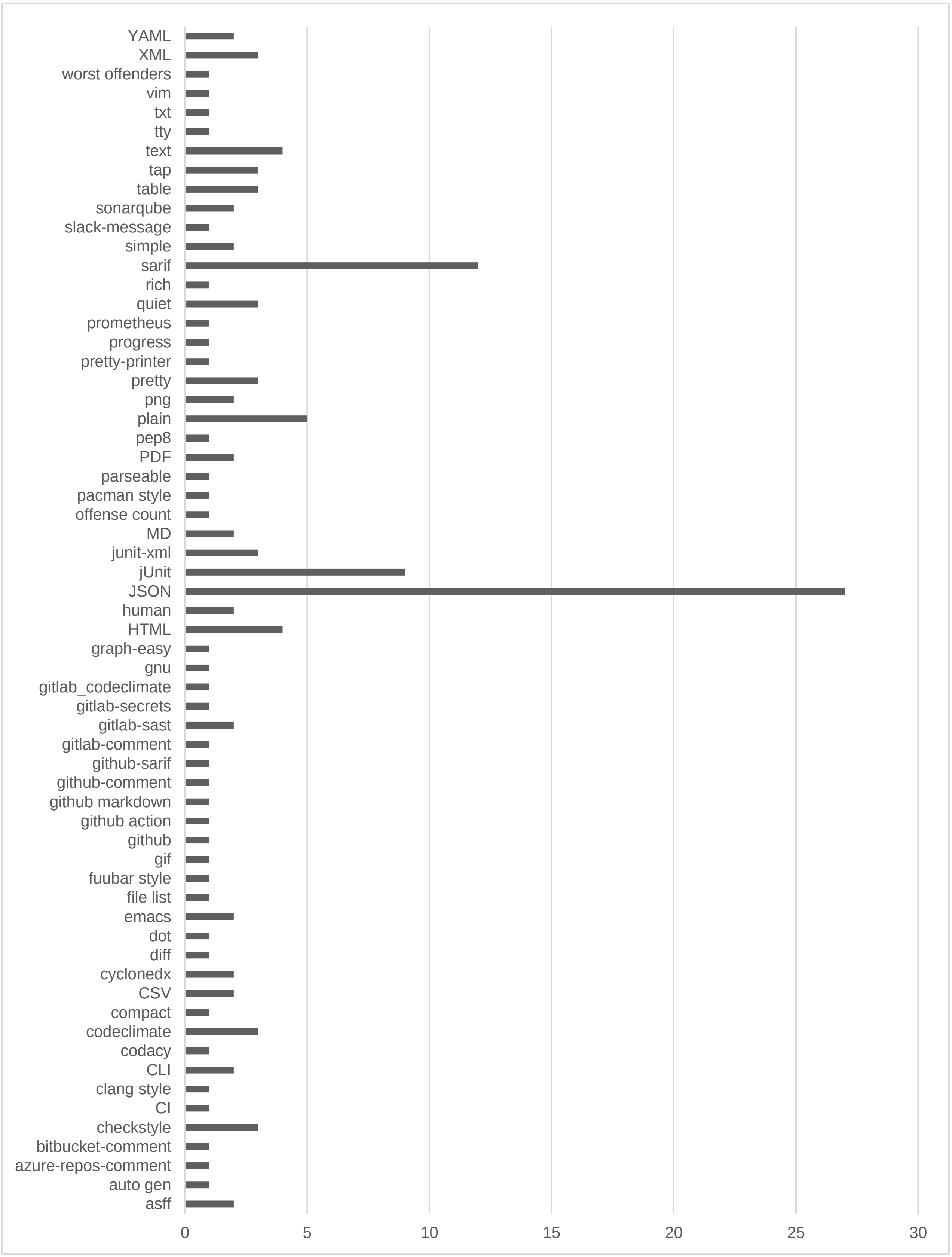
- Participants are confused about what happens if they select multiple options
- Users find it challenging to locate data in the detailed information section per SICA
- Users are confused why the detailed information section of some SICAs missed some headings
- Do not show “rule implementation” options when the option “custom checks” is not selected
- Improve resetting the search
- Better handling of custom IaC tools
- Statistics about the displayed SICAs
- Advanced sorting

All other findings were general ideas for improvements unrelated to one of the tasks.

Appendix E

Tool Output

Figure E-1. SICA Characteristics: Tool Output



Source: Own representation.

Appendix F

SICA Assessment

General information

The columns in Table F-1 are now shortly explained:

- **SICA**: The first column contains the **names of the SICAs** selected for the in-depth assessment.
- **Repository**: The repository column entails the **names of the repositories** against which the SICAs mentioned in the first column were tested.
- **IaC Tool**: **IaC tools of the repositories** the SICAs were tested against are in the third column. As explained earlier, *checkov* and *KICS* support multiple IaC tools. These SICAs are tested against the IaC tool for which they have the most rules. Furthermore, repositories may contain manifests of different IaC tools.
- **Findings**: The findings column **quantifies the findings** of a SICA in a particular repository. If the SICA assigns severity levels to the findings (e.g., warning), the findings are grouped by severity levels. Otherwise, one single number is given, which represents all findings.
- **Time**: The fifth column, “T (s),” shows a SICA's **average execution time (in seconds)** against a particular repository. The SICA was executed five times, and the average was calculated.
- **Cohen’s Kappa** Score: The values in the column “CKS” refer to the **practitioner’s classification into true and false positives**. When two practitioners agreed that all findings were true positives, Cohen’s Kappa score could not be calculated. Since the agreement between the two raters was **100%, it was set to 1**.
- **P1**: P1 shows the **true positive (TP) rate of a SICA’s findings** in a particular repository according to the first practitioner who classified the findings. Beyond that, the second number in the column represents the **repair rate (FIX)**, indicating how many findings the first practitioner would resolve.
- **P2**: P2 shows the **true positive (TP) rate of a SICA’s findings** in a particular repository according to the second practitioner who classified the findings. Beyond that, the second number in the column represents the **repair rate (FIX)**, indicating how many findings the second practitioner would resolve.

Table F-1. SICA Assessment

SICA	**Repository**	**IaC Tool**	**Findings**	**T (s)**	**CKS**	**P1: TP, FIX**	**P2: TP, FIX**
ansible-lint	*postgresql_cluster*	*Ansible*	Failure: 81 Warning: 1	39	1	100% 98%	100% 98%
	openshift-ansible	*Ansible*	47[16]	11:80	0	45% 45%	100% 100%
	CG: Monitoring VM	*Ansible*	Failure: 100 Warning: 2	12:40	0.837	72% 8%	54% 0%
	CG: Kubernetes setup	*Ansible*	Failure: 67 Warning: 29	7	-0.149	90% 90%	22% 22%
puppet-lint	*puppet-nginx*	*Puppet*	17	2	1	100% 0%	100% 6%
	puppet-os-hardening	*Puppet*	1	1:10	1	100% 100%	100% 100%
	~~*CG: no repository identified*~~	~~*Puppet*~~	-	-	-	-	-
	~~*CG: no repository identified*~~	~~*Puppet*~~	-	-	-	-	-
cookstyle	*aws*	*Chef*	18	4:10	1	100% 61%	100% 61%
	chef-cookbook	*Chef*	1	4:20	1	100% 100%	100% 100%
	~~*CG: no repository identified*~~	~~*Chef*~~	-	-	-	-	-
	~~*CG: no repository identified*~~	~~*Chef*~~	-	-	-	-	-
tfsec	*terragoat*	*Terraform*	Critical: 20 High: 86 Medium: 87	2:80	-0.042	96% 62%	96% 56%

[16] For this particular repository, no severity levels were noted during the evaluation. The analysis results could not be reproduced later.

SICA	Repository	IaC Tool	Findings	T (s)	CKS	P1: TP, FIX	P2: TP, FIX
			Low: 46				
	cloudblock	*Terraform*	Critical: 16 High: 6 Medium: 13 Low: 4	1:40	1	100% 21%	100% 33%
	CG: Product Configuration	*Terraform*	High: 40 Medium: 3 Low: 53	28	-0.079	58% 58%	96% 96%
	CG: Notification Service	*Terraform*	Critical: 27 High: 8 Low: 37	0:80	0	100% 76%	8% 8%
terrascan	*terragoat*	*Terraform*	High: 34 Medium: 28 Low: 5	5:50	1	100% 42%	100% 40%
	cloudblock	*Terraform*	High: 8 Medium: 8 Low: 3	8:80	1	100% 47%	100% 78%
	CG: Product Configuration	*Terraform*	High: 11 Medium: 60 Low: 5	21	0	100% 100%	98% 98%
	CG: Notification Service	*Terraform*	Critical: 9 High: 4 Low: 3	2:20	0.375	69% 69%	38% 25%
hadolint	*dockerfiles*	*Dockerfile*	Error: 253 Warning: 641 Info: 152 Style: 1	4:50	0	100% 100%	86% 86%
	Dockerfile	*Dockerfile*	Warning: 28 Info: 32 Style: 4	1:60	0	100% 100%	90% 90%

SICA	Repository	IaC Tool	Findings	T (s)	CKS	P1: TP, FIX	P2: TP, FIX
	CG: Confluence Synchronisation Tool	*Dockerfile*	1	0:20	1	100% 100%	100% 100%
	CG: Angular Web Application	*Dockerfile*	3	2	-0.8	33% 33%	67% 67%
cfn_nag	*cfngoat*	*CloudFormation*	Failure: 14 Warning: 22	1:20	1	100% 61%	100% 42%
	aws-cf-templates	*CloudFormation*	Failure: 79 Warning: 349	4:20	1	100% 42%	100% 72%
	~~*CG: no repository identified*~~	~~*CloudFormation*~~	-	-	-	-	-
	~~*CG: no repository identified*~~	~~*CloudFormation*~~	-	-	-	-	-
KubeLinter	*kubernetes-goat*	*Kubernetes*	103	0:80	1	100% 100%	100% 100%
	kubernetes-examples	*Kubernetes*	153	1:20	1	100% 100%	100% 88%
	CG: Monitoring services	*Kubernetes*	126	1:30	0.291	86% 84%	90% 24%
	CG: Development services	*Kubernetes*	268	8	0.087	94% 94%	42% 42%
checkov	*terragoat*	*Terraform*	345	8:10	0.059	92% 58%	66% 66%
	cloudblock	*Terraform*	104	14:10	-0.036	98% 64%	86% 72%
	CG: Product Configuration	*Terraform*	94	17	0	100% 100%	88% 88%
	CG: Notification Service	*Terraform*	43	6:30	0.164	91% 79%	47% 19%

SICA	**Repository**	**IaC Tool**	**Findings**	**T (s)**	**CKS**	**P1: TP, FIX**	**P2: TP, FIX**
KICS	*terragoat*	*Terraform*	High: 103 Medium: 63 Low: 32 Info: 62	43:20	0	92% 70%	100% 44%
	cloudblock	*Terraform*	High: 49 Medium: 18 Low: 17 Info: 433	57:10	1	98% 66%	98% 74%
	CG: Product Configuration	*Terraform*	High: 11 Medium: 43 Low: 3 Info: 21	155	-0.066	86% 86%	96% 96%
	CG: Notification Service	*Terraform*	High: 19 Medium: 4 Low: 9 Info: 81	33:10	-0.063	86% 86%	28% 0%

Source: Own representation.

Appendix G

YAML Schema

Listing G-1. YAML Schema

```
name: str()
toolSupport: any(list(str()), null())
fileSupport: any(list(str()), null())
builtIn: bool()
includedTools: any(list(str()), null())
categories: any(list(str()), null())
developmentSupport:
  ide: any(list(str()), null())
  ci: any(list(str()), null())
  vc: any(list(str()), null())
repository:
  url: any(str(), null())
  stars: any(num(), null())
  contributors: any(num(), null())
  license: any(str(), null())
  backers: any(str(), null())
release:
  firstRelease: any(str(), null())
  lastRelease: any(str(), null())
rules:
  builtInChecks: any(bool(), null())
  customChecks: any(bool(), null())
  blacklistChecks: any(bool(), null())
  ignoreFindings: any(bool(), null())
  whitelistChecks: any(bool(), null())
experiments: any(list(include('experiment')), null())
usage:
  documentation:
    quality: any(str(), null())
    link: any(str(), null())
  webApplication: any(str(), null())
  installation: any(list(str()), null())
  autoFix: any(bool(), null())
  output: any(list(str()), null())
implementation:
  languages: any(list(str()), null())
  ruleImplementation: any(list(str()), null())
---
experiment:
  name: str()
  executedBy: any(list(str()), null())
  dateOfExperiment: any(str(), null())
  truePositives: any(num(), null())
  falsePositives: any(num(), null())
  speed: any(num(), null())
  fixRate: any(num(), null())
```

Source: Own representation.

VII. Glossary

Natural Language Processing (NLP)	NLP is "an area of computer science that deals with methods to analyze, model, and understand human language" (Vajjala et al., 2020, chapter 1). It comprises information extraction, text summarization, question answering, and many other areas (Vajjala et al., 2020, chapter 1).
Machine Learning (ML)	ML "is an approach to … learn … complex patterns from … existing data and use these patterns to make … predictions on … unseen data" (Huyen, 2022, chapter 1).
Semantic Versioning (SemVer) – Major, Minor, and Patch Version	Semantic versioning comprises guidelines for assigning and determining version numbers for software applications. The schema for a version number is MAJOR.MINOR.PATCH. (Preston-Werner, n.d.) The major version increases for incompatible API changes, and backward compatible functional changes increase the minor version. The patch version increases for backward compatible bug fixes. (Preston-Werner, n.d.)
Cloud Development Kit (CDK)	A cloud development kit is "an open-source software development framework to define … cloud application resources using familiar programming languages" like Java or Go (Amazon Web Services, n.d.–c).

VIII. Declaration of Authenticity

I hereby declare that I have completed this Master's thesis on my own and without any additional external assistance. I have made use of only those sources and aids specified and I have listed all the sources from which I have extracted text and content. This thesis or parts thereof have never been presented to another examination board. I agree to a plagiarism check of my thesis via a plagiarism detection service.

Kuppenheim, October 2, 2022	[illegible]
Place, Date	Student signature