

https://researchcommons.waikato.ac.nz/

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

A Quality Framework for Infrastructure as Code in the Modern DevSecOps Era

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Doctor of Philosophy in Computer Science

at

The University of Waikato

by

Pandu Ranga Reddy Konala



THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2026

"Ever tried. Ever failed. No matter.
Try again. Fail again. Fail better."

Samuel Beckett

Abstract

Infrastructure as Code (IaC) is transforming how organisations deploy and manage their infrastructure through version-controlled scripts. While this creates opportunities for automation, it also introduces quality and security concerns. Many IaC analysis tools focus on one or two platforms and rely on pattern-based detection, but do not take advantage of the information contained within repository metadata and the relationships between repositories. A survey of current IaC analysis tools reveals shortcomings in their ability to conduct comprehensive quality assessments, allowing security smells to propagate through dependency chains and code reuse patterns without being identified.

This thesis presents a systematic process for assessing the quality of IaC scripts. It introduces a nine-category framework that extends the principles of traditional software quality to infrastructure code. The categories include aspects such as metadata and platform-specific characteristics that were not addressed in previous works. This work also utilises policy-based weighted mechanisms to allow organisations to place their own emphasis on the various categories when conducting assessments. Building on this framework, the research assesses security smells by utilising metadata-assisted methods for detecting vulnerabilities in the supply chain of dependencies. Our findings show that vulnerable dependency chains affect multiple downstream repositories. For platforms that lack fork metadata, similarity-based methods are employed to identify the paths through which security smells diffuse. Clone detection proved effective at tracing these diffusion paths across such platforms. This work also demonstrates the applicability of the framework to the evaluation of AI-generated infrastructure code. The results showed that 15 out of the 16 tested AI models produced code that scored below human-written repositories. However, structured prompts with quality and compliance integration enabled the top-performing AI models to produce code that exceeded human-written implementations in meeting industry benchmarks for security. Together, these findings support the usefulness of the proposed framework for the integration of quality assurance into modern DevSecOps workflow practices.

Acknowledgements

This journey has been the most challenging and rewarding experience of my life. I am grateful for this opportunity and honoured to have met exceptional individuals who have taught and supported me throughout.

First and foremost, I express my gratitude to the School of Graduate Research, Dr. Vimal Kumar (Head of Cybersecurity Researchers of Waikato - CROW), and the MBIE-funded Catalyst project "AI for Human-Centric Security" for funding my PhD. I am deeply grateful to my supervisors: Dr. Vimal Kumar, Prof. David Bainbridge, and Dr. Junaid Haseeb. This research would not have been possible without your unwavering encouragement, support, and guidance through both successes and setbacks. I am fortunate to have been mentored by such knowledgeable and respected individuals. Thank you for the invaluable knowledge and memorable journey. I also thank Dr. David Nichols for the interesting and engaging discussions we shared during lunchtime.

Beyond the research lab, I thank the Department of Computer Science for providing me the opportunity to work as a Doctoral Assistant. Special thanks to Dr. Annika Hinze, Dr. Colin Pilbrow, and Dr. Shaoqun Wu for their generous support, which allowed me to pursue this opportunity without financial concern. My e-Tuts colleagues have been a constant source of support. My heartfelt thanks go to Tracey Morgan, Dr. Brendan Sheridan, Dr. Sina Enteshari, Dr. Benjamin Dorrington Redder, Dr. Alea Ann Macam, Dr. Brenda Farias da Silva and many more colleagues at Te Puna Ako – Centre for Tertiary Teaching and Learning (CeTTL). Your support has been invaluable in maintaining my well-being throughout this degree. Our movie nights, board games, and shared laughter created some of my most joyful memories, reminding me that life exists beyond research.

Within the research environment, CROW has been instrumental to my success. To my colleagues Dr. Atthapan Daramas, Harpreet Kaur, Ralph Huey Olegario, Kevin Han, Bedir Asici, and Dr. Christopher Chew Jun Wen, thank you for creating such a supportive and welcoming environment. You provided me with a sense of belonging and kept me grounded throughout my research. Your talent and expertise have taught me immensely about both technology (NZCSC CTF, Hack The Box), history (EU4, CK3, Blackadder)

and the culture of New Zealand. Thank you for the unforgettable memories and encouragement during difficult times.

Distance never diminished the support of my close friends. I thank Sri Harsha Patallapalli, Arvind Sai, Sanjay, Sai Arvind, Vineeth, Vamsi Krishna Reddy, and many others who provided mental support from afar through gaming sessions in Dota 2, Arma 3, Command: Modern Operations, and Call of Duty, making this journey less isolating.

Finally, to my beloved family, who have been my foundation throughout this journey and my entire life. I honour the memory of my great-grandparents who are no longer with us. To my grandparents, thank you for your constant encouragement and positive spirit. Sravanthi, my wife, your encouragement and hard-working nature inspire me daily. A special thanks to Ganga Bhavani for your unwavering support and kindness throughout this journey. Rukmini, my mom, thank you for working tirelessly to ensure my well-being. Thammi, my dad, you have been my pillar through every trial, believing in me even when I doubted myself and going above and beyond to provide comfort during all the difficult times. You are my greatest inspiration and the reason I strive for excellence.

I dedicate this thesis to all who have offered unwavering support throughout the years. From the bottom of my heart, thank you!

Table of Contents

1 Introduction **1**

1.1 Understanding Infrastructure as Code 1
1.2 Code and Security Smells in Infrastructure Code 3
1.3 Problem Statement 5
1.4 Research Contributions 9
1.5 Thesis Structure 11

2 Background **15**

2.1 Background on IaC and its Analysis Tools 16
2.2 Research Method and Literature Selection 24
2.3 Current IaC Static Analysis Tools and Approaches 27
2.4 Current IaC Dynamic Analysis Tools and Approaches 30
2.5 Smell Detection Techniques in IaC 31
2.6 IaC Analysis Tool Landscape and Evolution 35
2.7 Discussion 42
2.8 Chapter Summary 45

3 An IaC Quality Framework **47**

3.1 Related Work 48
3.2 Deriving Quality Metrics from IaC Code Attributes 57
3.3 Quality Score Computation Algorithm 74
3.4 Chapter Summary 86

4 Policy-Driven Quality Assessment and Empirical Analysis **89**

4.1 The Role of Policy in Software Quality Assessment 90
4.2 Policy Implementation in the IaC Quality Framework 91
4.3 Attribute-Level Policy Configurations 94
4.4 Category-Level Policy Configurations 99
4.5 Evaluation Dataset 108
4.6 Attribute-Level Policy Evaluation and Results 112
4.7 Category-Level Policy Evaluation and Results 132
4.8 Discussion 145

4.9 Chapter Summary 150

5 Quality-Based Attack Detection 152
5.1 Related Work 153
5.2 IaC Tool Capabilities for Supply Chain Detection 157
5.3 Methodology 161
5.4 Evaluation and Results 167
5.5 Discussion 174
5.6 Chapter Summary 175

6 Quality-Based Pattern Analysis 176
6.1 Related Work 177
6.2 IaC Tool Capabilities for Diffusion Analysis 180
6.3 Methodology 182
6.4 Evaluation and Results 194
6.5 Discussion 200
6.6 Chapter Summary 203

7 Quality Assessment of AI-Generated IaC Scripts 204
7.1 Related Work 205
7.2 Baseline Evaluation of Foundation Models 215
7.3 Structured Prompting and Compliance Integration 223
7.4 Evaluation and Results 232
7.5 Discussion 244
7.6 Chapter Summary 253

8 Conclusion 255
8.1 Future Work 257

Appendices 274

A Supplementary Figures 275

B CO-STAR Quality Directives 282

List of Figures

1.1 Hierarchy of the research contributions and thesis chapters, showing the Chapter 3 framework, Chapter 4 analysis, and applications in Chapters 5–7 alongside the challenges addressed. . . . 12

2.1 Layered Architecture of IaC Tool Categories [1] 19
2.2 Temporal Evolution of Static Analysis Tools for IaC by Maturity Category (2011-2023) 38
2.3 Distribution of Static Analysis Tool Maturity Categories by Script Analysis Location 41

3.1 Three activities informing the derivation of the nine IaC quality categories. 62
3.2 Alignment of Proposed IaC Code Quality Framework with ISO/IEC 25010:2011 67

4.1 Organisation of Eight Development Philosophies Across Three Growth Categories 101
4.2 Six-Month Aggregate Statistics for Total Quality Scores Across Repository Dataset 114
4.3 Six-Month Aggregate Performance for Categories with Increasing Scores 116
4.4 Six-Month Aggregate Performance for Categories with Stable Scores 118
4.5 Six-Month Aggregate Performance for Categories with Declining Scores 119

4.6 Attribute-Level Policy Comparison: Equal-Weight vs Best-Practice Configurations 120
4.7 Total Quality Score Evolution Across Six Nginx Repositories . 123
4.8 Nine Quality Category Score Trajectories for Nginx Repositories 125
4.9 Average Quality Category Scores Across Nginx Repositories: Policy Configuration Comparison 127
4.10 Total Quality Scores by Nginx Repository: Equal-Weight vs Best-Practice Policy Comparison 128
4.11 Higher Scores Under Best-Practice Policy: `mrlesmithjr.elkstack` Repository 130
4.12 Lower Scores Under Best-Practice Configuration: `genzouw.selinux-disable` Repository 131
4.13 Category-Level Policy Alignment: RHEL 7 CIS Organisational and Individual Repository Comparison With Dataset Baseline 134
4.14 Organisational Repository Philosophy Evolution: Official Nginx Repository Across Three Development Stages (enlarged versions are provided in Appendix A, Figure A.1, Figure A.2, and Figure A.3) 138
4.15 Philosophy Alignment Deviations Across Development Stages: Official Nginx Repository 140
4.16 Individual Repository Philosophy Evolution: `geerlingguy.nginx` Repository Across Three Development Stages (enlarged versions are provided in Appendix A, Figure A.4, Figure A.5, and Figure A.6) 143
4.17 Philosophy Alignment Deviations Across Development Stages: Individual Nginx Repository 144

5.1 Analysis Granularity Levels in IaC Static Analysis Tools . . . 158
5.2 Exemplar Dependency Chains Extracted from IaC Script Dataset 167
5.3 Temporal Distribution of Repository Releases with Ansible Version Release Timeline 169

7.1 Structured Evaluation Protocol with Data Collection Across Multiple Stages 217
7.2 Sample CIS Level 1 (IG1) Control for Apache Tomcat 10 Benchmark [2] 227
7.3 Alignment of Level 1 CIS Critical Security Controls [3] with IaC Code Quality Framework Components 229
7.4 Changes in the nine category scores (0 to 1 each) from baseline to CO-STAR prompting under the best practice policy. Red cells mark decreases of 0.01 or more, and models are grouped by standards awareness. 236
7.5 Relationship between context window capacity and code quality scores under CO-STAR prompting. 238

A.1 Organisational Repository Philosophy Evolution: Official Nginx Repository, Stage 1 – Initial (enlarged from Figure 4.14) . . . 276
A.2 Organisational Repository Philosophy Evolution: Official Nginx Repository, Stage 2 – Development (enlarged from Figure 4.14) 277
A.3 Organisational Repository Philosophy Evolution: Official Nginx Repository, Stage 3 – Present (enlarged from Figure 4.14) . . . 278
A.4 Individual Repository Philosophy Evolution: `geerlingguy.nginx` Repository, Stage 1 – Initial (enlarged from Figure 4.16) . . . 279
A.5 Individual Repository Philosophy Evolution: `geerlingguy.nginx` Repository, Stage 2 – Development (enlarged from Figure 4.16) 280
A.6 Individual Repository Philosophy Evolution: `geerlingguy.nginx` Repository, Stage 3 – Present (enlarged from Figure 4.16) . . . 281

List of Tables

0.1 Mapping of publications to research contributions and thesis chapters. iii

2.1 Sources consulted in the survey. 25
2.2 Systematic Classification of Static Analysis Tools for IaC . . . 29
2.3 ML-Based Smell Detection Approaches for IaC 33
2.4 Mapping of the gaps identified in the review to research contributions and thesis chapters. 45

3.1 Overview of IaC Platforms and Associated Quality Metrics by Category 49
3.2 Comparison of Quality Scoring Systems across IaC Platforms . 49
3.3 Mapping of IaC Code Attributes from Rahman et al.'s [4] to ISO/IEC 25010:2011 Quality Characteristics 59
3.4 Mapping of Code Quality Attributes from Dalla Palma et al.'s [5] to ISO/IEC 25010:2011 Quality Characteristics 60
3.5 Common Quality Elements Across IaC Platform Documentation: Infrastructure Provisioning and Image Building 65
3.6 Common Quality Elements Across IaC Platform Documentation: Configuration Management 66
3.7 Mapping of IaC Attributes from Rahman et al.'s [4] to Proposed Quality Framework 70
3.8 Mapping of Iac Attributes from Dalla Palma et al.'s [5] to Proposed Quality Framework 71

3.9 Mapping of Code Attributes to Proposed Code Quality Framework Categories 77
3.10 Worked Example of Attribute Normalisation for the Values 10, 12, 15, 18, and 200 81
3.11 Comparison of Min–Max and Median-Based Normalisation for the Values 10, 12, 15, 18, and 200 83

4.1 Category Weights for Different Development Philosophy Approaches 105
4.2 Mapping of Code Attributes to Proposed Code Quality Framework Categories 112

5.1 Analysis Granularity and Metadata Processing Capabilities of IaC Static Analysis Tools 160
5.2 Repository Activity and Author Contribution Metrics in IaC Ecosystem 171
5.3 Number of Ansible Repositories Affected by Supply-Chain Vulnerabilities 171

6.1 Data Availability and Integration Features of IaC Distribution Platforms 181
6.2 Confusion Matrix and Performance Metrics for Pairwise Fork Detection Across Test Sets of Elasticsearch (3,916 pairs), Jenkins (2,556 pairs), and MySQL (2,556 pairs) 192
6.3 Analysis of Security Smell Diffusion Patterns Using Cumulative Common Weakness Enumeration Counts 196
6.4 Distribution of Cumulative Repository Downloads by Security Smell Diffusion Pattern Category 197

7.1 Comparative Analysis of AI-Enabled Development Tools and Command-Line Interfaces 206

7.2 Features and Benchmark Performance Comparison of Large Language Models 209
7.3 Relationship Between Compliance Frameworks and CIS/STIG Standards 212
7.4 Comparative Baseline Evaluation of AI Model Code Quality Performance by Policy Setting (Grouped by Category) 220
7.5 Category-Level Performance Analysis with Human and Dataset Averages 220
7.6 Manual Code Inspection Results Across AI Models 223
7.7 Dataset Composition and Compliance Requirements 232
7.8 Performance Ranking of AI Models Across Different Policy Criteria for Tomcat and MongoDB Technologies 233
7.9 AI Models Meeting Both Quality and Compliance Requirements 242

Nomenclature

AMI	Amazon Machine Images
ARM	Azure Resource Manager
AST	Abstract Syntax Trees
AWS	Amazon Web Services
CI/CD	Continuous Integration/Continuous Deployment
CIS	Center for Internet Security
CNN	Convolutional Neural Networks
CoT	Chain-of-Thought
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
HCL	HashiCorp Configuration Language
IaC	Infrastructure as Code
IDE	Integrated Development Environment
IG	Implementation Group
LLMs	Large Language Models
OPA	Open Policy Agent
SQuaRE	Software Quality Requirements and Evaluation
STIGs	Security Technical Implementation Guides
TOSCA	Topology and Orchestration Specification for Cloud Applications
ToT	Tree-of-Thoughts

List of Publications

The research work done for this thesis has resulted in the following publications:

- Part of the work presented in Chapter 2 has been published as **Konala, P.R.R.**, Kumar, V., and Bainbridge, D., "SoK: Static Configuration Analysis in Infrastructure as Code Scripts," in *2023 IEEE International Conference on Cyber Security and Resilience (CSR)*, Venice, Italy, 2023, pp. 281–288, https://doi.org/10.1109/CSR57506.2023.10224925.

- The work presented in Chapter 5 has been published as **Konala, P.R.R.**, Kumar, V., Bainbridge, D., and Haseeb, J., "Metadata Assisted Supply-Chain Attack Detection for Ansible," in Katsikas, S., Shafiq, B. (eds) *Data and Applications Security and Privacy XXXIX. DBSec 2025.* Lecture Notes in Computer Science, vol 15722. Springer, Cham. https://doi.org/10.1007/978-3-031-96590-6_18.

- The work presented in Chapter 6 has been published as **Konala, P.R.R.**, Kumar, V., Bainbridge, D., and Haseeb, J., "Tracking Security Smell Diffusion Patterns in Ansible Playbooks Using Metadata," in Cid, C., Yanai, N. (eds) *Advances in Information and Computer Security. IWSEC 2025.* Lecture Notes in Computer Science, vol 16208. Springer, Singapore. https://doi.org/10.1007/978-981-95-4674-9_19.

- The research presented in Chapters 3 and 4 has been submitted to a journal for publication.

- The research presented in Chapter 7 has been submitted to a conference for publication.

Table 0.1 maps each publication to the research contributions defined in Chapter 1 and the chapters presenting them, distinguishing published from in-progress work.

Table 0.1: Mapping of publications to research contributions and thesis chapters.

Publication	Venue	Status	Contribution	Chapter
SoK: Static Configuration Analysis in IaC Scripts	IEEE CSR 2023	Published	Foundation for RC1	2
Metadata Assisted Supply-Chain Attack Detection for Ansible	DBSec 2025	Published	RC2	5
Tracking Security Smell Diffusion Patterns in Ansible Playbooks Using Metadata	IWSEC 2025	Published	RC3	6

Chapter 1

Introduction

1.1 Understanding Infrastructure as Code

Infrastructure as Code (IaC) represents a transformation in computing resource and configuration management that has achieved substantial industry adoption and market growth. Market analyses by Precedence Research and Global Market Insights estimate the IaC market reached USD 850 million to USD 1.06 billion in 2024, with projections indicating expansion to USD 3.3–9.4 billion by 2030–2034, representing compound annual growth rates between 20% and 24%.[1][2] Industry surveys report that DevOps adoption reached 74–83% of organisations by 2024,[3][4] reflecting industry-wide transition from manual infrastructure configuration to programmatic management approaches. Organisations now define, provision, and manage infrastructure through machine-readable definition files rather than manual processes or interactive configuration tools [1]. This transformation enables version control of infrastruc-

[1]Precedence Research, "Infrastructure as Code Market Size to Reach USD 9.40 Bn by 2034," 2024, https://www.precedenceresearch.com/infrastructure-as-code-market.

[2]Global Market Insights, "Infrastructure as Code Market Size and Share: Growth Report 2034," 2024, https://www.gminsights.com/industry-analysis/infrastructure-as-code-market.

[3]Paul Krill, "Most Developers Have Adopted DevOps, Survey Says," InfoWorld, April 2024, https://www.infoworld.com/article/2337172/most-developers-have-adopted-devops-survey-says.html.

[4]Bryan Reynolds, "The State of DevOps in 2025: Trends, Adoption, Challenges, and Future Directions," Bay Tech Consulting, May 2025, https://www.baytechconsulting.com/blog/the-state-of-devops-in-2025.

ture specifications, automated deployment workflows, and configuration consistency across development, testing, and production environments.

The transition from manual processes to using code to manage and configure infrastructure is largely driven by scale. As infrastructure grows, it exceeds the ability of humans to manually perform all necessary work. Manual configuration processes also introduce inconsistencies, create bottlenecks within deployment pipelines, and do not allow for full reproducibility of infrastructure. IaC addresses these limitations by treating and automating the organisation's infrastructure as source code, making it subject to standard software engineering practices such as peer review, testing, and continuous integration. The IaC ecosystem comprises three types of tool categories, each addressing a different aspect of automation [1]: *Infrastructure provisioning* creates and manages cloud-based resources through either declarative or imperative specification; *Configuration management* automates the installation and configuration of software applications onto provisioned infrastructure; and *Image building* constructs images that contain pre-configured software and applications for use in deployments. When organisations use these tools at scale, they introduce quality and security issues similar to those found in traditional software development environments.

However, there are key differences in how infrastructure code operates compared to traditional application code. *State management* in IaC refers to tracking provisioned resources and reconciling the desired specification of what should be provisioned against what has actually been provisioned, in order to determine whether resources need to be created, modified, or removed. Unlike application code, which runs transiently, infrastructure code interacts with persistent external state [6]. This interaction demands *idempotence*, which ensures that running the same script repeatedly produces the same outcome. Idempotence is also essential for continuous deployment pipelines; without it, duplicate resources can be created, configuration drift can occur, or pipeline executions can fail [6]. Furthermore, because IaC scripts operate with *elevated*

privileges, they can make changes to system configurations, install software, and grant access permissions. If an IaC script contains a misconfiguration at this privilege level, it could lead to unauthorised access. A single vulnerability in an IaC script can therefore compromise an entire system, whereas a similar vulnerability in application code typically impacts only a specific function.

Understanding the types of patterns that contribute to poor-quality infrastructure code, introduce security vulnerabilities, and create maintenance problems across various IaC implementations is therefore important. These patterns are referred to as code smells and security smells. The following sections describe these smells in detail, along with the challenges associated with detecting and remediating them.

1.2 Code and Security Smells in Infrastructure Code

Code smells represent problems in software design that make code difficult to maintain or evolve [7]. *Security smells* are a subset of code smells that represent patterns which may contain vulnerabilities [8]. Each smell type has unique characteristics in IaC environments. Code smells manifest as issues such as inadequate error checking and non-idempotent operations, while security smells manifest as hard-coded passwords and access control misconfigurations.

Such patterns in infrastructure code pose risks beyond traditional software, as they directly control system configurations and access permissions. The prevalence and impact of these smells in production environments have demonstrated operational and financial consequences. StrongDM reports that infrastructure misconfigurations account for 15% of initial attack vectors in security breaches.[5] The Thales Group Cloud Security Study found that human error and misconfigurations constitute 31% of cloud breach root causes, with

[5]StrongDM, "40+ Alarming Cloud Security Statistics for 2025," March 2024, https://www.strongdm.com/blog/cloud-security-statistics.

44% of organisations experiencing cloud data breaches.[6] Tenable surveyed organisations on cloud security and documented that 95% suffered cloud-related breaches over an 18-month period.[7] Check Point and Cybersecurity Insiders reported that 61% of enterprises experienced security incidents attributed to cloud misconfigurations.[8]

Analysis of IaC scripts reveals the technical basis for these incidents. Palo Alto Networks Unit 42 conducted a security analysis of cloud infrastructure configurations and found that 42% of CloudFormation and 22% of Terraform files contained at least one insecure configuration.[9] Separately, Jit Security analysed public GitHub repositories for exposed credentials and detected 10 million hard-coded secrets in commits during 2022, representing a 67% increase from the previous year.[10] These security incidents translate to direct financial impact. IBM Security's Cost of a Data Breach Report found that the global average cost of a data breach reached USD 4.4 million in 2025.[11] StrongDM reports that breaches attributed to cloud misconfigurations cost organisations an average of USD 3.86 million.[12] This evidence demonstrates that code and security smells in infrastructure code represent quality issues with financial consequences, requiring detection and remediation approaches.

Current detection tools provide mechanisms to identify code and security issues in IaC scripts through methods such as regular expressions, graph analysis, and formal verification [8, 9, 10]. However, existing tools target specific platforms and smell categories. For example, SLIC [8] analyses configuration

[6]Thales Group, "2024 Thales Cloud Security Study," June 2024, https://www.infosecurity-magazine.com/news/cloud-breaches-half-organizations/.

[7]Tenable, "2024 Cloud Security Outlook," 2024, https://www.tenable.com/cyber-exposure/2024-cloud-security-outlook.

[8]Check Point Software Technologies and Cybersecurity Insiders, "2024 Cloud Security Report," April 2024, https://www.checkpoint.com/resources/items/cloud-security-report-2024.

[9]Palo Alto Networks Unit 42, "Unit 42 Cloud Threat Report: Spring 2020," February 2020, https://unit42.paloaltonetworks.com/cloud-threat-report-intro/.

[10]Jit Security, "Hard-Coded Secrets in GitHub Repositories: 2022 Analysis," 2022, https://www.jit.io/resources/appsec-tools/top-10-infrastructure-as-code-security-tools-for-2024.

[11]IBM Security, "2025 Cost of a Data Breach Report," 2025, https://www.ibm.com/think/x-force/2025-cost-of-a-data-breach-navigating-ai.

[12]StrongDM, "40+ Alarming Cloud Security Statistics for 2025," March 2024, https://www.strongdm.com/blog/cloud-security-statistics.

management scripts using pattern matching, while Checkov[13] and KICS[14] examine infrastructure provisioning templates using a checklist. These differing detection approaches mean organisations employing multiple IaC technologies must use separate tools, resulting in fragmented coverage and inconsistent methodologies.

1.3 Problem Statement

The treatment of IaC quality in existing work has been piecemeal. Individual tools and studies each address a small subset of quality concerns, such as a particular platform, a specific family of security smells, or a narrow set of maintainability metrics. Each applies its own detection methodology and reports findings in its own format and severity conventions, so results are inconsistent and difficult to compare across tools. Research effort has concentrated on detecting individual issues, and a prior question has received little attention: what does quality mean for an IaC codebase considered as a whole? We are not aware of any existing work that offers a comprehensive account of IaC quality across platforms and concerns. This thesis addresses that gap. Its aim is comprehensiveness, meaning the breadth of quality concerns organised within a single framework, and it does not seek complete detection coverage of every issue on every platform.

This inconsistency has a deeper source. No agreed definition of IaC quality assessment exists, and the tools reflect that absence. Prior work has instead produced taxonomies of what can go wrong. Rahman et al. proposed the Seven Sins [8], defining seven categories of security smell in configuration management scripts, and later the Gang of Eight [11], defining eight categories of defect derived from qualitative analysis of defect-related commits. These two taxonomies gave the field a shared vocabulary, and subsequent academic work

[13]Bridgecrew by Palo Alto Networks, "Checkov," 2019, https://www.checkov.io/.

[14]Checkmarx, "KICS: Keeping Infrastructure as Code Secure," 2021, https://www.kics.io/.

on smell detection builds on the categories they introduced. A taxonomy, however, answers a different question from the one an assessment must answer. It states that a finding belongs to a category, yet says nothing about how serious that category is relative to the others, how a set of findings combines into a single judgement about a repository, or which categories should matter most in a given organisational context. Nor does it look beyond the contents of the script, so repository metadata such as dependency relationships and maintenance history falls outside its scope. A taxonomy is therefore a backbone for assessment rather than an instrument of it, and the framework developed in this thesis builds on these taxonomies rather than competing with them.

General software quality standards provide the assessment structure that these taxonomies lack, but they are not written with infrastructure code in mind. ISO/IEC 25010 [12] does not include characteristics and definitions relevant to the concerns of infrastructure code. State management, idempotency requirements, and platform-specific configuration are infrastructure-related concerns that existing quality models do not address. Without tailored metrics for evaluating IaC quality, organisations are unable to establish a common baseline and track changes over time, making it difficult for teams to evaluate the relative quality of different IaC projects. The original edition of the standard, ISO/IEC 25010:2011 [12], was the current version when the analytical work in this thesis commenced. This edition was subsequently superseded by ISO/IEC 25010:2023 [13], which was published during the course of this research. The standard follows a revision cycle of approximately a decade, with ISO/IEC 25010:2011 having itself replaced ISO/IEC 9126-1:2001 [14] after a ten-year interval. The 2023 revision introduces a safety characteristic and applies terminology refinements, renaming “usability” to “interaction capability” and “portability” to “flexibility.” These revisions do not alter the structural decomposition of quality into characteristics and sub-characteristics, nor do they introduce definitions that address infrastructure code concerns such as state management, idempotency, or repository metadata. The mapping and

analysis presented in this thesis therefore use the 2011 edition.

While IaC tools exist to analyse codebases, current analysis methodologies suffer from metadata blindness. These methodologies focus on properties at the code level but fail to consider important repository characteristics. For example, metadata related to dependency versions, dependency relationships, and maintenance history provide signals about potential vulnerabilities that can help improve the overall security of the system. Because current analysis methodologies [11, 15] do not use the information contained in repository metadata, they miss vulnerabilities that early detection and remediation could have prevented , thereby increasing the risk of supply chain attacks.

In the IaC context, such an attack exploits the dependency chain itself. A widely used role may depend on a third-party component whose maintainer has abandoned it or unintentionally introduced a vulnerability, or an adversary may deliberately plant an obfuscated vulnerability in a dependency, impersonate a popular component through typosquatting, or compromise a maintainer account. In every case, the payload executes with the elevated privileges of each downstream deployment that installs it. Documented cases show that this pathway is real. In March 2025, attackers compromised tj-actions/changed-files, an automation component used by more than 23,000 repositories, retroactively re-pointing its version tags at a malicious commit that exfiltrated CI/CD secrets into public build logs (CVE-2025-30066).[15] In 2026, a flaw disclosed in ansible-core (CVE-2026-11332) demonstrated that the dependency specification in a role's `meta/requirements.yml` file can itself be an attack vector, allowing a malicious role author to inject git configuration flags through the `src` field and achieve arbitrary code execution on the machine of any user who installs the role.[16] Chapter 5 examines the supply-chain threat landscape and these mechanisms in detail.

[15]National Vulnerability Database, "CVE-2025-30066," March 2025, https://nvd.nist.gov/vuln/detail/CVE-2025-30066; GitHub Advisory Database, "GHSA-mrrh-fwg8-r2c3," March 2025, https://github.com/advisories/GHSA-mrrh-fwg8-r2c3.

[16]National Vulnerability Database, "CVE-2026-11332," June 2026, https://nvd.nist.gov/vuln/detail/CVE-2026-11332.

This problem is further compounded when we take repository relationships into consideration. Vulnerabilities may be located across different IaC platforms that do not retain metadata regarding forks, making them difficult to identify. A common practice among software development practitioners includes code reuse, which involves taking code developed during one project and utilising it in order to speed up development of another project. The code reuse strategy provides an additional pathway for vulnerabilities to remain hidden, since when this reused code is cloned from the original source, it will contain the same security smells as the original source. Because security smells propagate from the original source to its clones, vulnerabilities become amplified across ecosystems [16]. Unfortunately, current platforms and tools used to perform IaC security assessments do not provide mechanisms to detect and track relationships among repositories, so teams performing security assessments lack visibility into the patterns of vulnerability diffusion that affect their IaC systems.

These existing challenges intensify with the emergence of AI-powered coding assistants, which introduce quality concerns at scale. Foundation models generate infrastructure code by replicating patterns from training data, which may include code and security smells present in the training corpus. The deployment velocity of AI-generated code surpasses manual review capacity, risking systematic quality degradation.[17][18] Harness conducted a survey on AI adoption in software engineering and found that while 63% of organisations report shipping code faster since adopting AI assistants, 45% of deployments involving AI-generated code lead to problems, and 72% of organisations have experienced production incidents attributed to AI-generated code.[17] For IaC specifically, these quality concerns compound existing risks. Given the elevated privileges discussed earlier, deficiencies in infrastructure scripts can result in

[17]Harness, "The State of AI in Software Engineering 2025," September 2025, https://www.harness.io/the-state-of-ai-in-software-engineering.

[18]Bill Harding and Matthew Kloster, "Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality," GitClear, 2024, https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality.

system-wide vulnerabilities regardless of authorship. AI-generated infrastructure code inherits these risks at scale and velocity that outpace manual review capacity.

To address these collective challenges, this thesis develops a quality assessment framework for IaC. The research is motivated by the observation that current software quality frameworks are unable to assess infrastructure-specific issues, and therefore a quality assessment framework specifically designed for IaC is needed. To validate that the framework identifies concerns not present in previous software quality frameworks, the research utilises the metadata features of the framework to perform supply chain vulnerability detection and determine whether the metadata provides sufficient information to identify vulnerabilities that are undetectable using currently available code-based tooling. A second validation is performed using the metadata and security categories of the framework to trace the propagation of security smells through cloned repository relationships and to determine whether the framework can identify security-related concerns that require multiple layers of evaluation. As the emergence of AI-assisted development environments that use foundational AI models for generating code creates uncertainty regarding output quality, this thesis investigates the application of the framework to evaluate AI-generated infrastructure code and determine whether AI models can produce secure baseline code by default. Based on the results of this investigation, a zero-shot method of addressing quality deficiencies prior to adding compliance requirements is developed. This method investigates whether capable AI models can generate production-ready infrastructure code that meets all security requirements.

1.4 Research Contributions

The following contributions emerge from this research:

- **Research Contribution 1:** *IaC Quality Framework and Policy-Driven*

Quality Assessment - This contribution introduces a nine-category framework that extends traditional software quality principles with additional categories for infrastructure-code-specific attributes, including metadata. It also presents mechanisms using policies to quantitatively assess quality based on these categories. Testing demonstrates its applicability across multiple repositories and confirms that it supports organisational customisation through configurable policies.

- **Research Contribution 2:** *Metadata-Assisted Supply Chain Attack Detection* - This contribution proposes a method to detect supply chain vulnerabilities through metadata analysis. The method uses the quality framework to analyse dependency chains, maintenance patterns, and repository characteristics that are not utilised by standard IaC code analysis tools. It leverages metadata indicators (e.g., author activity, update frequency, dependency relationships) to identify previously unknown vulnerability pathways in IaC ecosystems.

- **Research Contribution 3:** *Security Smell Diffusion Analysis* - This contribution studies how security vulnerabilities diffuse across repository relationships in platforms where no fork metadata is available. The proposed methodology combines the quality framework's ability to detect security smells with similarity-based clone detection algorithms to identify repository relationships and trace the diffusion patterns of security smells. The study evaluates code overlap behaviours, categorises diffusion patterns, and explores the impact that platform metadata gaps have on the spread of undetected vulnerabilities.

- **Research Contribution 4:** *Code Quality Assessment of AI-Generated IaC Scripts* - This contribution evaluates 16 AI models producing 278 Ansible roles and develops recommendations for improving the quality of AI-generated IaC scripts. The evaluation assesses whether AI models produce secure IaC code by default and compares the results against their

respective dataset baselines. The evaluation of the CO-STAR prompting framework examines whether providing structured instructions improves the quality of generated IaC scripts. Finally, compliance integration is studied to determine which models are capable of generating CIS-compliant IaC scripts, establishing the foundation for quality-aware AI code generation to support the deployment of production-ready code.

Together, these contributions address the need for comprehensive assessment of IaC. The framework in Chapter 3 establishes how quality is measured, while Chapter 4 examines how organisational priorities affect that assessment and the patterns observed across repositories. Once quality can be measured, the remaining contributions examine where the IaC code that practitioners deploy comes from. This thesis examines three routes through which practitioners obtain IaC code that they did not write themselves, namely depending on another repository, copying code from another repository, and generating code with an AI model. Chapters 5, 6, and 7 apply the framework to these routes in turn, examining supply-chain vulnerabilities introduced through dependencies, security smell diffusion through copied code, and the quality of AI-generated IaC scripts. Figure 1.1 maps the challenges identified in Section 1.3 to these contributions and the chapters that present them.

1.5 Thesis Structure

This thesis is structured around the following chapters . Figure 1.1 summarises how these chapters relate to the identified challenges and research contributions. Rather than a single literature review chapter, each chapter includes relevant background and prior work that contributes to a broader understanding of the existing body of research.

Chapter 2 lays the groundwork by surveying the range of analysis tools available for Infrastructure as Code. This chapter reviews 24 tools, classified based on two criteria: whether they were produced within an academic

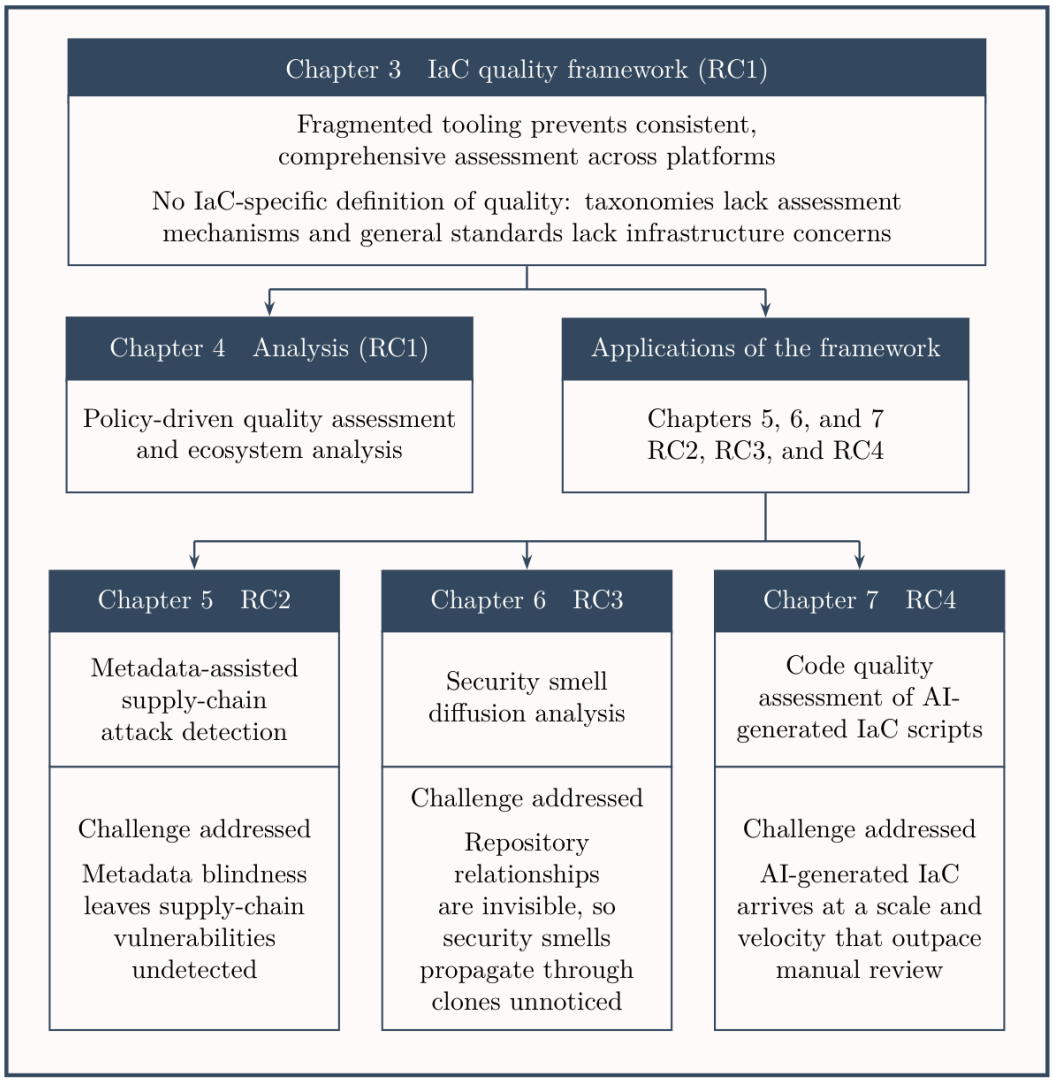


Figure 1.1: Hierarchy of the research contributions and thesis chapters, showing the Chapter 3 framework, Chapter 4 analysis, and applications in Chapters 5–7 alongside the challenges addressed.

or industrial environment, and the type of IaC platform the support is provided. The conclusion of this chapter is that there is a fragmentation in detection approaches and platform coverage among IaC configuration analysis tools. Additionally, through a systematic examination of detection techniques and maturity levels, the review identifies gaps in current methodologies for assessing IaC scripts. Finally, the chapter discusses a number of empirical studies that have documented IaC quality issues relevant to the present study.

Chapter 3 presents the core contribution of this thesis which is a quality assessment framework for IaC. The chapter introduces a computational methodology that expands upon traditional software quality models to account for infrastructure code. The chapter describes nine specialised categories for assessing IaC quality characteristics, identified through mapping exercises with ISO/IEC 25010:2011. These categories reflect attributes unique to infrastructure code and highlight where traditional software quality characteristics fall short when applied to IaC.

Chapter 4 builds on the quality assessment framework by introducing policy-based mechanisms for evaluating IaC quality. The chapter focuses on enabling organizations to define their priorities as policies, which are subsequently evaluated using quality metrics. It explains how these policies convert generic quality metrics into assessments that are specifically tailored to the organization's context. The chapter uses empirical analysis of large-scale Ansible Galaxy repositories to demonstrate the framework's practical utility. It shows how different policy configurations can reveal patterns in quality evolution and identify opportunities for organizational alignment.

Chapter 5 applies the quality assessment framework to perform security analysis of IaC configurations, with a particular focus on identifying supply chain vulnerabilities. The contribution of this chapter is the development of metadata-assisted algorithms capable of detecting vulnerable dependencies in IaC roles that are invisible to traditional static code analysis. The chapter reviews research related to common attack vectors and methodologies for de-

tecting supply chain attacks. The investigation demonstrates that metadata indicators such as maintenance patterns and author activity correlate with the presence of vulnerabilities in IaC configurations.

Chapter 6 investigates security smell diffusion patterns across repository relationships using the quality assessment framework. The primary goal of this chapter is to address the challenge of tracking vulnerability propagation in platforms that do not maintain fork metadata. The chapter develops similarity metrics based on clone detection techniques adapted from traditional software engineering for use in IaC contexts, and uses them to identify repository relationships. The investigation demonstrates that security smells propagate through code reuse in IaC, showcasing hidden vulnerability pathways that were not previously discovered in the IaC domain.

Chapter 7 evaluates the utility of the quality assessment framework through the assessment of AI-generated infrastructure code. The chapter begins by reviewing the current state-of-the-art foundation models and their code generation capabilities, followed by the application of the framework to evaluate the output of 16 AI models. The investigation develops strategies for improving the quality of AI-generated code, including the use of structured prompting optimised for quality and compliance. The chapter also discusses how practitioners can leverage AI assistance while still producing high-quality code suitable for deployment in production environments.

Chapter 8 provides an overview of the contributions of this thesis and demonstrates how the quality assessment framework enables both practical security discovery and secure development workflows. The chapter discusses the progression from the development of the framework through its security-related applications to its use as a tool for practitioners, highlighting its ability to address a range of challenges associated with assessing IaC quality. The chapter concludes by identifying future research areas at the intersection of quality measurement, security analysis, and automated code generation.

Chapter 2

Background[1]

IaC represents a paradigm shift in system administration where infrastructure management occurs through machine-readable definition files rather than manual configuration processes or interactive configuration tools. This transition brings software engineering practices to infrastructure management, enabling version control, peer review, and automated deployment. However, it also introduces quality concerns emanating from traditional software development such as code smells. Code smells indicate potential design problems that complicate maintenance or extension of the codebase [7]. Security smells constitute a category of code smells representing characteristics that suggest vulnerabilities [8]. Both smell types in IaC scripts can lead to misconfigurations, causing infrastructure failures and security breaches at scale.

Detecting these issues before deployment requires systematic analysis techniques that can examine scripts without executing them. Such techniques have evolved alongside IaC adoption, with both academic and industry efforts contributing various tools and approaches to address quality concerns. This chapter investigates the current landscape of IaC analysis by first establishing foundational concepts of how IaC tools are categorised and how scripts are structured. With this foundation in place, the investigation then examines existing analysis tools and their detection techniques to understand what

[1]This chapter largely builds upon P. R. R. Konala et al.'s research [17], incorporating revisions for thesis presentation.

capabilities currently exist. Finally, the analysis considers tool maturity and evolution patterns to identify where gaps remain and what opportunities exist for advancement.

2.1 Background on IaC and its Analysis Tools

In order to understand how configuration analysis tools can be used with IaC, we need to evaluate both the historical development of configuration management and the different types of IaC Tools that have developed over time. This will first involve reviewing how IaC has transitioned from manually configuring infrastructure, through automation using code, and then we will look at the use of analysis tools for IaC.

2.1.1 Evolution of IaC

IaC evolved from shell scripting to code-based automation because of DevOps practices [1]. As infrastructure automation continued to grow and as concerns were being divided up into layers of the infrastructure stack, three distinct groups of tools emerged, each of which is meant to manage one layer of the infrastructure stack.

Infrastructure Provisioning operates at the foundational layer of infrastructure management. This category addresses the challenge of creating and managing cloud resources programmatically. Infrastructure provisioning tools define infrastructure components using either imperative or declarative code. As a result, these tools bring the same level of process rigour to infrastructure (such as version control, peer review, and repeatable deployments) as is used in application development.

Terraform exemplifies this category through its provider-based architecture, which supports multiple cloud providers as well as on-premises infrastructure. Terraform uses HashiCorp Configuration Language (HCL) to define the desired state of infrastructure and maintains a state file that tracks resource

configurations and dependencies. *TOSCA* (Topology and Orchestration Specification for Cloud Applications) defines a standard language for describing and orchestrating the topology of cloud applications, enabling portability across cloud providers. Cloudify extends this approach by using TOSCA-defined blueprints as the basis for workflow automation, managing the entire lifecycle of cloud applications.

Rather than manually performing console operations, which can lead to configuration drift between environments, these tools enable reproducible environments and consistency between development and production. If an incident occurs, teams can recreate infrastructure directly from code rather than relying on memory or incomplete documentation. Additionally, the abstraction layers provided by these tools support multi-cloud strategies and reduce vendor lock-in.

Configuration Management addresses the need for consistent software configurations across large-scale server fleets after they have been provisioned. While infrastructure provisioning creates the resources, configuration management ensures those resources maintain the correct software versions, configurations, and desired states over time. This category also addresses "configuration drift," which occurs when servers deviate from their desired state due to manual changes, failed updates, or environmental factors.

Configuration management tools use different models to achieve the desired state. Puppet utilises a declarative model in which the administrator defines the desired end state and the tool determines the actions needed to reach it. Puppet's master-agent architecture enables centralised configuration management, with agents periodically checking in with the Puppet master to ensure compliance. Chef uses an imperative model through Ruby-based recipes and cookbooks, offering flexibility in how configuration logic is defined. Chef's client-server architecture includes a Chef server for centralised management, a workstation for authoring configurations, and Chef clients installed on the managed nodes. Ansible differentiates itself through an agentless architecture,

using SSH to communicate with Linux/Unix systems and WinRM for Windows systems. By removing the need for agents, Ansible reduces operational overhead and simplifies adoption in existing environments.

Configuration management addresses real-world operational challenges that manual methods cannot. Maintaining consistency across hundreds or thousands of servers is impractical without automation, as is enforcing uniform security policies and compliance standards. These tools eliminate the burden of repetitive tasks such as user account management and package updates, and enable automatic configuration of new infrastructure as part of continuous deployment pipelines.

Image Building is another approach to IaC that focuses on building immutable, pre-configured infrastructure components. Unlike the other categories, image building addresses the variability introduced when servers are configured at runtime, which can extend deployment times. Pre-configured machine images contain operating systems, software packages, and configurations, enabling rapid and consistent deployment.

HashiCorp Packer automates the creation of images across multiple platforms and supports formats including Amazon Machine Images (AMI), Docker containers, VMware templates, and Azure images. Packer uses JSON or HashiCorp Configuration Language (HCL) templates specify how images should be built, including provisioners and post-processors to produce identical images across platforms. Configuration management tools such as Chef, Puppet, and Ansible can be integrated into the build process to configure images prior to finalisation.

Docker has advanced image building significantly through containerisation, though its scope extends well beyond traditional machine images. Dockerfiles describe file system changes in layers, resulting in lightweight, portable images. The layered approach allows for efficient storage and transfer of layers across images. Additionally, the build process produces immutable artifacts that ensure consistency throughout the application lifecycle, from development

through to production.

The primary advantage of image building is the significant reduction in time between provisioning and service availability. Since every instance is launched from an identical image, the variability associated with runtime configuration is eliminated. Security teams benefit from the ability to produce pre-hardened images prior to deployment. The versioned nature of images also enables advanced deployment strategies such as blue-green releases, where traffic is shifted between image versions rather than requiring in-place updates.

Category Interactions and Integration: In practice, the categories described above form a layered architecture (as shown in Figure 2.1). Infrastructure provisioning creates the base layer, configuration management maintains the desired state of an organisation's software resources, and image building produces pre-configured components. Most organisations use a combination of tools from across these categories to create a comprehensive automation pipeline.

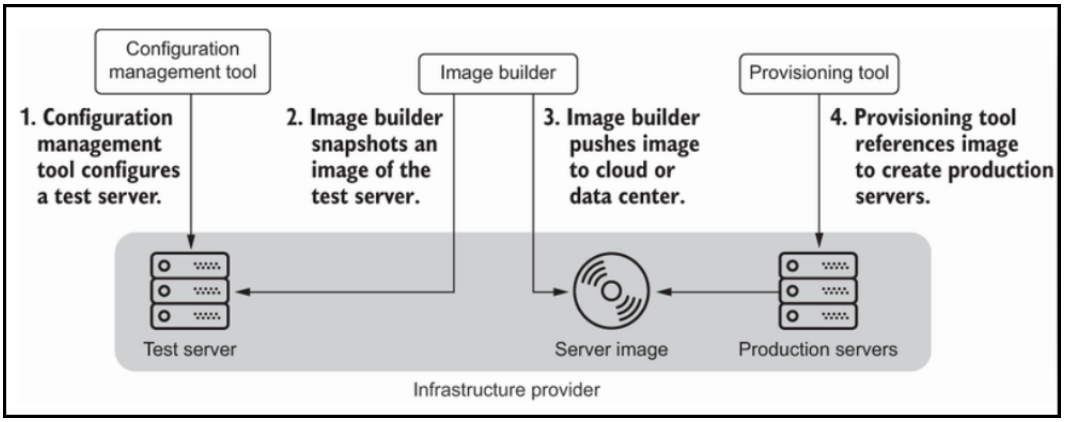


Figure 2.1: Layered Architecture of IaC Tool Categories [1]

A common example of this layered approach is as follows. Terraform provisions the cloud-based infrastructure for an application. Packer then builds machine images with the necessary configurations embedded. Once the images have been created, Ansible performs post-deployment configuration. This layered approach is effective because each tool is optimised for a specific task: Terraform for infrastructure abstraction, Packer for consistent image creation,

and Ansible for flexible machine configuration.

The lines between the layers in this hierarchy are increasingly being blurred by newer implementations of the various categories of IaC. For example, Terraform has introduced a new function of provisioner which can be used to run configuration management type scripts. Also, Ansible provides modules for use in provisioning infrastructure and Docker Compose provides orchestration services for containerised applications. Overall, there appears to be a trend towards developing all-encompassing platform IaC solutions for automating infrastructure, but recognising the need for specialised tools based on application-specific needs also continues.

In order to identify configuration errors, it is necessary to understand how the different categories of IaC interact with one another. Each category has its own set of code smells and security risks. For example, scripts designed to provision infrastructure may include sensitive information (e.g., credentials) or incorrectly configured security group settings. Scripts used to manage configuration may have hardcoded password values or set file permissions that are too permissive. Image definitions used to build images may include known vulnerabilities or outdated software versions. To find configuration errors across multiple categories of IaC, it will be necessary to review the underlying structures of the scripts that make up those configurations.

2.1.2 Understanding IaC Scripts

The defining features of every IaC script are attributes, which serve as a fundamental component of all configuration. Each attribute is represented by a key-value pair that defines the configuration for system placement, permissions and management. Attributes allow organisations to automate the configuration and management of their infrastructure and software.

IaC can use either individual scripts or repositories of multiple scripts to accomplish infrastructure automation tasks. Scripts may target a single system or operate across multiple systems within distributed infrastructures. The

following Ansible playbook snippet [18] demonstrates the use of attributes in a practice, as shown in Listing 2.1.2.

Listing 2.1.2: Sample Ansible Script Illustrating Key-Value Attribute Components

```
1: - name: Install Nginx
2:   hosts: web_servers
3:   become: yes
4:   tasks:
5:     - name: Install Nginx
6:       apt:
7:         name: nginx
8:         state: present
```

Each attribute performs a specific function according to Ansible documentation specifications [2]:

- The `name` attribute specifies the purpose of the play, defined as "Install Nginx."
- `hosts` designates target machines for task execution, referencing a group labeled `web_servers`.
- `become` enables privilege escalation, allowing task execution with elevated permissions.
- `tasks` lists actions to perform on target servers, with each entry representing a task for Ansible execution.
- `apt` specifies the module for managing packages on Debian-based systems.
- `state` within the `apt` module defines the desired package state. Setting `state: present` ensures Nginx installation if absent from the system.

Infrastructure teams distribute IaC scripts through specialised IaC codebases such as Ansible Galaxy, Chef Supermarket or generic version control platforms including GitHub. These platforms extract metadata from designated files within repositories. Ansible Galaxy obtains metadata from the

[2]Michael DeHaan and Red Hat, Inc., "Ansible," 2012, https://www.ansible.com/.

`meta/main.yml` file present in each role repository. Script retrieval through command-line interfaces includes metadata acquisition that precedes script download. The following example illustrates metadata structure obtained from Ansible Galaxy [3] as shown in Listing 2.1.2:

Listing 2.1.2: Sample Metadata Schema for Ansible Galaxy Role

```
{ "id": 56789,
  "created": "2024-02-05",
  "username": "devops_admin",
  "github_repo": "ansible-role-nginx",
  "github_user": "devops_admin",
  "github_branch": "master",
  "name": "nginx","summary_fields": {
    "dependencies": [{"id": 10,
    "name": "sec.hardening"},"..."],
    "namespace": {"id": 3050,
    "name": "devops_admin"},
    "provider_namespace": {"repository":
    {"name": "nginx"},
    "tags": ["webserver","nginx"],
    "versions": [{"name":"1.1"}, "..."]}},
    "downloads": 342560 }
```

Metadata provides critical information for infrastructure automation management. Dependencies indicate required components for script execution, enabling compatibility verification before deployment. Platform specifications ensure script applicability to target environments. Version information facilitates change tracking and rollback capabilities. Download metrics and star ratings indicate adoption levels and community validation. The metadata structure supports script discovery, compatibility assessment within the IaC ecosystem.

The structured nature of IaC scripts, characterised by defined attributes and metadata, enables systematic examination through automated techniques. The key-value pair structure provides identifiable patterns for analysis algorithms, while metadata offers contextual information for assessing script characteristics. This structural foundation, combined with the increasing complexity of infrastructure configurations and the distribution of scripts across multiple repositories, makes manual review impractical at scale where auto-

[3] Red Hat, Inc., "Ansible Galaxy," 2016, https://galaxy.ansible.com/.

mated approaches to quality assessment become essential.

2.1.3 Code Analysis Approaches for IaC

Two code analysis approaches have emerged to address the automation need for IaC quality assessment. Each approach leverages the structural properties of IaC scripts to detect issues, though with distinct capabilities and constraints.

- **Static Analysis** examines configuration scripts without requiring execution, using syntactical, structural and pattern-based techniques. Configuration scripts can be analysed statically through manual expert review or through automated tools that parse the syntactical structure of scripts and apply a set of predefined detection rules. A key benefit of static analysis is that infrastructure does not need to be provisioned and deployed for testing purposes, reducing both the cost and time associated with evaluating configuration scripts. Additionally, configuration-related risks can be identified and remediated early, prior to production deployment and at a point when remediation costs remain low.
- **Dynamic Analysis** evaluates infrastructure behaviour by executing scripts in test environments. This process provisions infrastructure components, executes them, and monitors runtime behaviour to discover issues that may arise during execution, such as configuration drift, resource conflicts, and timing dependencies. These issues cannot be detected through static analysis alone, as no execution environment is available. Dynamic analysis provides assurance that infrastructure components are functioning correctly in terms of both functional and operational characteristics, that resources are being properly provisioned, that dependency relationships are being resolved, and that the system is operating within acceptable performance boundaries. However, dynamic analysis requires test infrastructure to be available, which introduces additional time and cost constraints that do not exist with static analysis methods.

The two approaches complement each other rather than serving as alterna-

tives. Static methods detect syntactic and structural issues without execution overhead while dynamic methods validate runtime behaviour. Integration of both approaches provides coverage across the defect lifecycle from code authorship through operational deployment. The remainder of this chapter surveys how existing research and tools address these analysis requirements, beginning with the method used to identify them.

2.2 Research Method and Literature Selection

To establish the current state of IaC research against these requirements, we conducted a structured survey combining academic literature and industry tools. We organised the survey around four questions: what research and tools currently exist for the configuration analysis of IaC scripts, what techniques these tools employ to detect code and security smells, what insights can be gained from the categories of IaC targeted by existing tools, and which specific aspects or characteristics of IaC existing tools address. Structuring the review around explicit questions rather than an open-ended reading of the literature kept the selection and analysis focused on the capabilities and limitations of current tooling. We conducted the survey as a systematisation of knowledge rather than as a systematic literature review, and we report the sources, search terms, selection criteria, and screening steps in full so that the review can be reproduced and extended.

Sources and search terms: Table 2.1 lists the sources we consulted. We used IEEE Xplore, SpringerLink, and the ACM Digital Library as the primary databases because they index the software engineering venues in which IaC research appears. Supplementary academic sources captured preprints and cross-indexed work, and we included grey-literature sources because a substantial share of IaC tooling is developed in industry without accompanying peer-reviewed publications. This deliberate inclusion of tools that are used and deployed without peer-review distinguishes our survey from the earlier

systematisation by Chiari et al. [19], which considered only tools supported by peer-reviewed publications and therefore excluded widely used solutions. The search covered the period from 2011 to 2023, corresponding to the emergence and growth of IaC practices. We organised the search terms in three concept groups and combined them as shown in Listing 2.2, adapting the syntax to each source.

Listing 2.2: Search Terms Used Across All Sources

```
IaC terms:
  ("infrastructure as code" OR "IaC" OR "configuration
  ↪ management"
   OR "infrastructure provisioning" OR "image building"
   OR "software automation")
AND ( Quality terms:
  ("security smells" OR "code smells" OR "defects"
   OR "misconfiguration" OR "errors" OR "anti-patterns"
   OR "vulnerabilities" OR "quality issues" OR "bad
   ↪ practices"
   OR "cyber security incidents" OR "quality assurance"
   OR "linting" OR "static analysis" OR "dynamic
   ↪ analysis")
AND Study-type terms:
  ("taxonomy" OR "classification" OR "categorisation" OR
  ↪ "survey") )
```

Table 2.1: Sources consulted in the survey.

Source	**Type**	**Role in the survey**
IEEE Xplore, ACM Digital Library, SpringerLink	Primary databases	Peer-reviewed publications on IaC analysis tools, techniques, and empirical studies
Google Scholar, ScienceDirect, arXiv, ResearchGate	Supplementary academic	Preprints and publications not indexed by the primary databases
GitHub, practitioner blogs, DevOps community resources	Grey literature	Industry tools in practical use without an accompanying publication

Selection criteria: We retained candidate work when it satisfied at least one inclusion criterion and none of the exclusion criteria.

- *Inclusion:* (I1) a tool that performs static or dynamic analysis of IaC scripts; (I2) a technique for detecting code smells or security smells in

IaC; (I3) an empirical study of IaC quality; (I4) an industry tool in wide practical use, with public documentation, that has no accompanying publication.

- *Exclusion:* (E1) work on general DevOps, CI/CD, or cloud operations that is not expressed through IaC scripts; (E2) work in which the acronym IaC or the term configuration carries an unrelated meaning; (E3) work not written in English; (E4) tools without a public release or documentation; (E5) duplicate reports of the same tool or study, of which the primary publication was retained.

Screening and analysis: We screened in three steps. We assessed titles and abstracts against the criteria, read the full text of the retained candidates, and followed the reference lists of the retained empirical studies to identify tools and studies that the searches had not surfaced. We reviewed the empirical studies first because they map the foundational research in the field, and their findings guided the identification of the tools and techniques examined in the remainder of the survey. We did not retain intermediate counts at each step, which is the principal limitation of the method. This process yielded six empirical studies of IaC quality and 24 static analysis tools, of which 17 are supported by publications and 7 are industry tools without formal publications. We synthesised the empirical studies narratively by theme (development and practitioner practices, smell and defect taxonomies, and secondary syntheses of research themes and practices). For the tools, we extracted target properties, detection techniques, availability models, DevOps integration capabilities, script analysis location, and supported IaC platforms, and classified each tool against these parameters to enable systematic comparison, systematised by detection technique, and categorised by maturity level based on availability and DevOps integration.

2.3 Current IaC Static Analysis Tools and Approaches

The survey revealed two main categories of contributions in the literature. The first category comprises empirical studies that analyse IaC script characteristics through repository mining, interviews, and systematic reviews. The second category comprises tools that detect specific problems in IaC scripts. This section examines both categories to establish the current knowledge base.

2.3.1 Review of Empirical IaC Studies

Empirical research on IaC script quality includes primary studies such as repository mining and practitioner interviews, and secondary studies including systematic mappings and literature reviews that synthesise broader evidence across the field.

Jiang and Adams [20] examined changes made to IaC scripts across multiple repositories. Their analysis revealed that regular developers, rather than infrastructure specialists, frequently modify these scripts. This pattern correlates with increased code quality issues and security misconfigurations. The study established the need for automated detection methods to address risks introduced by non-specialist modifications.

Guerriero et al. [21] conducted 44 interviews with IT professionals to understand IaC development practices. The research identified Ansible as the most adopted configuration management tool among practitioners. The study revealed gaps between available tools and practitioner needs, particularly in security smell detection.

Sharma et al. [22] analysed 4,621 Puppet repositories containing 142,662 files and 8.9 million lines of code. The analysis discovered co-occurrence patterns among security smell categories, suggesting that certain smell types cluster together. This finding indicates that detecting one smell type might signal the presence of others.

Rahman et al. [23] conducted a systematic mapping study and, after screening a large research corpus, identified 32 IaC-related publications grouped into key research themes including frameworks and tools, adoption, empirical studies, and testing. Their findings suggest that frameworks and tools constitute a well-studied topic in IaC research, while defects and security flaws remain under-explored despite their serious consequences for deployment and development environments in DevOps.

Kumara et al. [24] complemented academic evidence with a systematic literature review, organising practitioner-reported best and bad practices across IaC languages such as Ansible, Puppet, and Chef. Their findings reveal insights concerning best practices adopted by practitioners to address common code challenges, while evidencing that the field of IaC development, security and maintenance remains in its infancy and deserves further attention.

Chiari et al. [19] reviewed static configuration analysis tools documented in academic publications. However, it should be noted that their exclusion of tools without formal publications limited the scope of their analysis. This approach overlooks tools used in industry that lack academic documentation.

2.3.2 Categorising IaC Analysis Tools

Following the empirical understanding of IaC quality issues, numerous tools emerged to address identified problems. The classification of these tools reveals patterns in approach and coverage.

The survey identified 24 tools for static configuration analysis in IaC scripts: 17 supported by publications and 7 utilised in industry. Table 2.2 presents the classification of these tools based on multiple characteristics, where $\star$ = *Tools adopted by IT Industry* distinguishes industry-adopted tools from those primarily documented in academic literature.

The classification reveals patterns in tool development and deployment. Tools target different aspects of IaC, ranging from anti-patterns to security vulnerabilities, while detection techniques span pattern matching to machine

Table 2.2: Systematic Classification of Static Analysis Tools for IaC

Tool	Year	Target Prop.	Detection Tech.	Avail.	DevOp Int.	Analysis Loc.	Supported Platform
ACID [11]	2020	Anti-Patterns	Regex	OS	No	Onsite	Puppet
BARREL [25]	2018	Deploy. Plan Corr.	Graph Analysis	OS	No	On & Off	TOSCA
⋆ Checkov [26]	2021	Security Smells	Regex	Hybrid	Yes	Onsite	Infra. Prov.
⋆ CloudSploit [27]	2020	Security Smells	Unknown	Prop.	Yes	Offsite	Infra. Prov.
CookStyle [28]	2016	Code Smells	Regex	OS	No	Onsite	Chef
DeepIaC [9]	2020	Anti-Patterns	Deep Learning	OS	No	Unknown	Ansible
Foodcritic [29]	2011	Code Smells	Regex	OS	No	Onsite	Chef
GLITCH [15]	2022	Security Smells	Regex	OS	No	Onsite	Ansible
Häyhä [30]	2021	Security Smells	Graph Analysis	OS	No	Onsite	AWS CF
⋆ KICS [31]	2020	Security Smells	Regex	OS	Yes	On & Off	Multiple
Puppeteer [22]	2016	Code Smells	Regex	OS	No	Onsite	Puppet
RADON [32]	2021	Anti-Patterns	Data Mining	OS	No	Onsite	Ansible
Rehearsal [10]	2016	Code Smells	SMT Solvers	OS	No	On & Off	Puppet
SecGuru [33]	2014	Network Policies	SMT Solvers	Prop.	No	Unknown	MS Azure
SecureCode [34]	2020	Code Smells	Regex	Prop.	No	Unknown	Ansible
⋆ Semgrep [35]	2020	Security Smells	Regex	OS	Yes	Offsite	Multiple
SLAC [36]	2021	Security Smells	Regex	OS	No	Onsite	Ansible, Chef
SLIC [8]	2019	Security Smells	Regex	OS	No	Onsite	Puppet
⋆ SNYK IaC [37]	2023	Security Smells	AI Engine	Prop.	Yes	Offsite	Multiple
SODALITE [38]	2020	Multiple Prop.	Ontology	OS	Yes	Onsite	TOSCA
Sommelier [39]	2017	Deploy. Plan Corr.	Adhoc Alg.	OS	No	Onsite	TOSCA
⋆ SonarQube [40]	2016	Code & Sec. Smells	Regex	Hybrid	Yes	Offsite	Multiple
TAMA [41]	2022	Pattern Instances	Pattern Alg.	OS	No	Onsite	Ansible
⋆ tfsec [42]	2019	Security Smells	Regex	OS	Yes	Onsite	Infra. Prov.

Abbreviations: ⋆ = Tools adopted by IT Industry; Target Prop. = Target Property; Detection Tech. = Detection Technique; Avail. = Availability; DevOps Int. = DevOps Integration; Analysis Loc. = Script Analysis Location; OS = Opensource; Prop. = Proprietary; Regex = Regular Expression; Deploy. = Deployment; Corr. = Correction; Infra. Prov. = Infrastructure Provisioning; On & Off = Onsite & Offsite; AWS CF = AWS CloudFormation; MS = Microsoft; Alg. = Algorithms; Sec. = Security

learning approaches. Section 2.5 provides detailed examination of each detection technique category, analysing the strengths and limitations of regex-based, machine learning, graph-based, and formal verification approaches employed by these tools.

2.4 Current IaC Dynamic Analysis Tools and Approaches

Dynamic security smell analysis techniques are well-established in traditional software security research. Newsome and Song [43] introduced TaintCheck, demonstrating dynamic taint analysis for automatic detection of buffer overflow and format string exploits in C programs through runtime binary instrumentation. More recently, Sutter et al.[44] conducted a systematic literature review identifying 43 publications on dynamic security smell analysis for Android, encompassing network monitoring, system-call tracing, and dynamic taint analysis techniques. These approaches share a common methodology of executing code in controlled environments to observe runtime behaviour and detect vulnerabilities that static analysis cannot identify.

To the best of our knowledge, the NCC Group report[4] is the only work on dynamic security analysis specifically for IaC identified in our review. The report describes initial experiments in which IaC configurations are executed in mock cloud provider environments to observe runtime behaviour. Unlike the C and Android studies discussed above, this work directly examines IaC execution. Rapid7[5] provides general guidance on dynamic application security testing (DAST) rather than an IaC-specific approach.

The limited adoption of dynamic analysis for IaC stems from execution complexity, as dynamic approaches require provisioning actual or simulated

[4]NCC Group, "Testing Infrastructure as Code Using Dynamic Tooling," 2022, https://www.nccgroup.com/research-blog/testing-infrastructure-as-code-using-dynamic-tooling/.

[5]Rapid7, "Dynamic Application Security Testing (DAST)," 2024, https://www.rapid7.com/fundamentals/dast/.

cloud infrastructure, incurring costs for compute instances, storage, and network components during each test execution. Additionally, IaC scripts exhibit declarative semantics where the "runtime" spans entire infrastructure lifecycles rather than discrete program executions. These constraints make static analysis the predominant approach in practice. Static analysis techniques are easier to implement, test, and validate while requiring fewer resources and less specialised expertise than dynamic alternatives. Organisations can integrate static analysis into existing development workflows without provisioning infrastructure or managing runtime environments. Given this predominance, understanding the detection approaches employed by existing static analysis tools provides the foundation for evaluating current capabilities and identifying opportunities for enhancement.

2.5 Smell Detection Techniques in IaC

The variety of detection techniques employed by the existing tools reflects different approaches to identifying IaC quality issues. This section systematically examines each technique category, analysing strengths, limitations, and adoption patterns.

2.5.1 Regex-Based Pattern Matching Approaches

Pattern matching through regular expressions emerged as a popular approach, employed by 13 of the 24 analysed tools. This prevalence stems from implementation simplicity and customisation flexibility, factors that warrant detailed examination.

Rahman et al. [8] developed SLAC and SLIC using regular expressions to detect security smells in Ansible, Chef, and Puppet scripts. Their subsequent tool, ACID [11], extends this approach by examining git commit history through regular expression rules applied to commit message differences. This technique identifies patterns in version control that indicate security smell in-

troduction.

GLITCH [15], developed by Saavedra et al., converts scripts from multiple configuration management tools into *AST*s before applying regular expressions. This intermediate representation enables cross-platform smell detection. Comparative analysis showed GLITCH achieving 8 % points higher precision and 13 % points higher recall than SLAC and SLIC.

Code smell detection through regular expressions appears in Foodcritic [29] and Puppeteer [22], targeting Chef and Puppet scripts respectively. These tools identify issues including improper code alignment, long statements, and misplaced attributes. Cookstyle [28] evolved from this work, focusing on Chef cookbook quality through automated detection and correction of style, syntax, and logic issues.

SecureCode [34] addresses security risks from embedded shell scripts in configuration management tools. The tool integrates with third-party analysers like Shellcheck [6] and PSScriptAnalyzer [7] within CI/CD pipelines. Analysis of 1,492 Ansible scripts from 45 GitHub repositories yielded 3,535 issue detections with 116 false positives.

Industry tools leverage regular expressions through pattern rules for smell detection. KICS [31] integrates with GitLab [8] CI/CD pipelines for pre-deployment security checking. Semgrep [35] provides similar capabilities across multiple IaC platforms. Checkov [26] and tfsec [42] target cloud infrastructure provisioning scripts, particularly Terraform, using Software Composition Analysis for CVE detection in configuration scripts and container images.

2.5.2 ML Approaches for Smell Detection

While regular expressions are used widely among current tools, machine learning techniques represent an emerging approach for smell detection, attempting to achieve improved accuracy and adaptability over pattern matching. How-

[6] Vidar Holen, "ShellCheck," https://github.com/koalaman/shellcheck.
[7] Microsoft, "PSScriptAnalyzer," https://github.com/PowerShell/PSScriptAnalyzer.
[8] GitLab, "About the one DevOps Platform," https://about.gitlab.com/.

ever, adoption remains limited due to challenges. DeepIaC [9] employs CNNs to detect anti-patterns in Ansible scripts. The tool's training utilised the dataset from Rahman et al. [8], augmented with artificially introduced bugs to address data scarcity. Detection accuracy ranges from 79% to 92% for artificial bugs.

RADON [32] implements a defect prediction framework using data mining techniques. The tool processes 108 features related to IaC code and software metrics through various algorithms including feature selection, data balancing, normalisation, and classification. The authors evaluated multiple classification algorithms and found that Isolation Forest, an unsupervised anomaly detection algorithm, achieved optimal performance with 86% precision and 77% recall.

SNYK IaC [37] mentions that it incorporates artificial intelligence for smell detection and remediation suggestions. The proprietary nature of the tool prevents detailed analysis of algorithms and training data. This limitation highlights challenges in evaluating commercial machine learning implementations. Table 2.3 summarises the characteristics of these approaches.

Table 2.3: ML-Based Smell Detection Approaches for IaC

Tool	Year	Target Property	Technique and Training	Reported Performance	Platform
DeepIaC [9]	2020	Anti-Patterns	CNN trained on the dataset of Rahman et al. [8] augmented with artificially introduced bugs	79–92% accuracy on artificial bugs	Ansible
RADON [32]	2021	Anti-Patterns	Defect prediction over 108 code and software metric features, with Isolation Forest performing best	86% precision, 77% recall	Ansible
SNYK IaC [37]	2023	Security Smells	Proprietary AI engine, algorithms and training data undisclosed	Not disclosed	Multiple

2.5.3 Graph-Based and Formal Verification Approaches

Graph-based and formal verification approaches provide rigorous analysis capabilities for specific IaC aspects. These techniques offer theoretical soundness but require specialised expertise for implementation and use.

Graph analysis provides structural understanding of IaC configurations. BARREL [25] validates TOSCA management plans through graph representation of node templates. The tool creates state representations of infrastructure, validating plans against predefined labelled paths.

Häyhä [30] detects intra-update sniping vulnerabilities specific to IaC. These vulnerabilities arise from absent resource upgrade ordering. The tool converts CloudFormation templates into dataflow graphs, visualizing vulnerabilities through graphical interfaces.

SMT solvers enable formal verification of IaC scripts. Rehearsal [10] uses the Z3 SMT solver to identify Puppet script violations of determinism and idempotency principles. The tool converts scripts to formal language specifications for SMT analysis. Testing on 13 Puppet scripts detected smells in 6 instances.

SecGuru [33] applies Z3 SMT solving to network access control policies in Microsoft Azure environments. The tool encodes firewall actions and policies as SMT problems, determining packet filtering decisions through constraint solving. Integration with Azure cloud demonstrated practical deployment of formal methods.

2.5.4 Additional Detection Techniques

Beyond the main detection categories, some tools employ specialised techniques tailored to specific analysis requirements. These approaches demonstrate the diversity of methods applicable to IaC quality assessment. SODALITE [38] combines multiple approaches for TOSCA script analysis: ontologies for code smell detection, SPARQL and RDF triple stores for contextual analysis, and regular expressions for security smells. The tool includes an IDE

using ASTs for graphical TOSCA script design.

Sommelier [39] uses ad hoc algorithms to verify entity connections in TOSCA scripts, preventing undefined references from poor coding practices. TAMA [41] applies pattern algorithms to classify Ansible script bugs into security bugs and performance bugs. TAMA also identifies three testing patterns that correlate with bug appearance, namely assertion roulette, local only testing, and remote mystery guest. Analysis of 104 open source software repositories found that 45.2% contained at least one script with bugs.

These diverse detection techniques reflect varying levels of sophistication and maturity in IaC analysis tooling. Regular expression-based approaches dominate current practice, employed by 13 of the 24 analysed tools, due to their implementation simplicity and customisation flexibility. Machine learning and formal verification methods offer greater analytical capability but face adoption barriers including data scarcity and implementation complexity. The distribution of techniques across tools indicates that the field continues to rely on established methods while gradually incorporating advanced approaches. However, detection capability alone does not determine practical utility.

2.6 IaC Analysis Tool Landscape and Evolution

Understanding how detection techniques translate into production-ready tools requires examination of tool maturity and temporal evolution. Maturity assessment identifies readiness for production deployment, reveals adoption barriers, and informs future tool design. This section examines tool progression through multiple analytical lenses to understand the current state and identify development patterns.

2.6.1 Classifying IaC Tools by Maturity Level

The maturity of static analysis tools reflects the field's development stage and readiness for production deployment. Classification based on integration and availability characteristics from Table 2.2 reveals distinct maturity levels that indicate how tools transition from research prototypes to production-ready solutions.

The classification framework employs two dimensions: DevOps integration capability and availability model. DevOps integration indicates whether tools can be embedded in CI/CD pipelines, a requirement for modern software development. The availability model (open source, proprietary, or hybrid) reflects the tool's development philosophy and business model, influencing adoption patterns and evolution trajectories.

Tool maturity analysis reveals the development stage of IaC static analysis through five distinct categories:

- **C1: Open source without DevOps integration (14 tools)** - Research-focused implementations prioritizing algorithmic innovation over production features. These tools demonstrate novel detection techniques but lack the engineering necessary for production deployment.

- **C2: Proprietary without DevOps integration (2 tools)** - Commercial tools targeting stand-alone analysis scenarios. The proprietary nature suggests commercial viability perception, but absent DevOps integration limits adoption in automated workflows.

- **C3: Hybrid with DevOps integration (2 tools)** - Tools offering both open-source and commercial versions with pipeline integration. This model balances community engagement with revenue generation while meeting production requirements.

- **C4: Open source with DevOps integration (4 tools)** - Mature open-source tools ready for production deployment. These represent

successful transitions from research to practice, maintaining open development while meeting enterprise requirements.

- **C5: Proprietary with DevOps integration (2 tools)** - Commercial tools fully integrated into development workflows. These represent the highest maturity level, combining production readiness with sustainable business models.

The concentration of 14 tools in C1 suggests patterns about the field's current state. First, it may indicate active research interest in IaC, with academics and open-source contributors exploring diverse detection approaches. Second, it appears to highlight a gap between research outputs and production requirements, as most tools lack the engineering investment needed for DevOps integration. Third, it suggests opportunity for tool developers, as proven techniques from C1 tools could be engineered into higher maturity categories.

The limited presence in C4 and C5 categories (6 tools combined) suggests the field remains nascent from a production perspective. Among the mature tools that do exist, specialisation predominates, with each focusing on specific problem domains rather than comprehensive infrastructure analysis. The analysis suggests organisations would benefit from tools capable of detecting multiple smell categories across their entire infrastructure stack, yet current solutions address only narrow subsets of quality concerns. This scarcity creates market opportunity for tool developers who can bridge the research-to-production gap.

The distribution also reveals different development philosophies. Open-source tools (18 total) dominate the landscape, reflecting the community-driven nature of DevOps tooling. However, the emergence of proprietary tools (4 total) and hybrid tools (2 total) indicates growing commercial interest as IaC adoption expands.

2.6.2 Tool Evolution and Development Timeline

The chronological development of tools provides perspective on field growth and external influences shaping tool creation. Figure 2.2 illustrates the temporal distribution of tool releases across maturity categories. We correlated these release patterns with external events to identify factors that drove tool development.

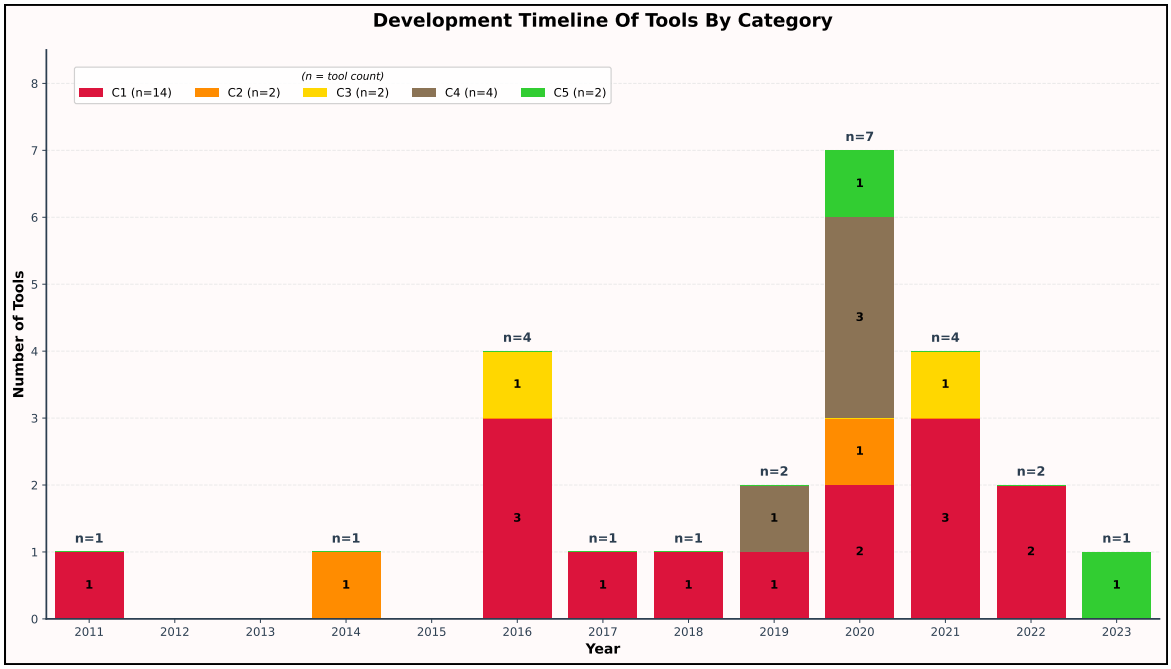


Figure 2.2: Temporal Evolution of Static Analysis Tools for IaC by Maturity Category (2011-2023)

We divide the chronological history of IaC tool development into three phases based on analysis of tool release dates from 2011 to 2023 and the correlation between tool creation and external events such as technology shifts and increased infrastructure automation demand.

Initial Phase (2011-2015): The emergence of foundational tools like Foodcritic [29] and SecGuru [33] marked the recognition that IaC scripts require quality assurance. This period saw limited tool development (2 tools), reflecting early IaC adoption and unclear quality requirements. Tools in this phase addressed basic syntax checking and pattern detection.

Growth Phase (2016-2019): Increased IaC adoption drove tool proliferation (8 tools), including Puppeteer [22], SLIC [8], and Rehearsal [10]. This

phase introduced sophisticated techniques like SMT solvers and began addressing specific security concerns. The emergence of cloud-native architectures and DevOps practices created demand for automated quality assurance.

Acceleration Phase (2020-2023): The COVID-19 pandemic catalysed dramatic growth (14 tools), representing 58% of all analysed tools. The pandemic drove digital transformation initiatives, forcing organisations to rapidly adopt cloud infrastructure and automation [19]. This urgency exposed IaC quality issues at scale, spurring tool development. Tools from this period incorporate diverse techniques ranging from regular expressions in ACID [11] and GLITCH [15] to deep learning in DeepIaC [9].

The progression from C1 to higher categories over time reveals maturation patterns. Early tools predominantly occupied C1, focusing on proving concepts. Recent years show increased C4 and C5 representation, indicating production adoption. However, most new tools still emerge in C1, suggesting continued research innovation alongside production engineering.

The timeline also reveals technology adoption curves. Regular expression-based tools appeared early and continue emerging, indicating their enduring relevance and ease of implementation. Machine learning approaches emerged recently (2020 onwards), reflecting both technology maturation and data availability improvements. Formal methods appeared mid-timeline but show limited growth, suggesting adoption challenges despite theoretical advantages. Complementing this temporal analysis, examining where tools integrate within development workflows provides additional indicators of production maturity.

2.6.3 Tool Integration Points in Development Workflows

The location where tools perform analysis reflects deployment models and architectural decisions that directly impact their practical utility. These integration patterns indicate tool sophistication, security considerations, and production readiness. Understanding analysis location helps organisations assess

tool suitability for their security and compliance requirements.

Script analysis location provides insights into tool deployment models and architectural evolution. Analysis occurs in four contexts, each with distinct implications:

Onsite Analysis: Tools performing local analysis maintain data within organisational boundaries, addressing security and compliance concerns. This approach suits organisations with strict data residency requirements or sensitive infrastructure configurations. However, onsite analysis requires local computational resources and tool maintenance.

Offsite Analysis: Cloud-based analysis enables scalability and eliminates maintenance overhead. Tools can leverage cloud computing resources for complex analysis and maintain updated smell databases. However, this requires transmitting potentially sensitive configuration scripts to external services, raising security concerns.

Both (Flexible): Tools supporting both modes provide deployment flexibility, allowing organisations to choose based on their requirements. This approach maximises adoption potential but increases development complexity.

Unknown: Proprietary tools often obscure analysis location, complicating security assessments. This opacity may hinder adoption in security-conscious organisations.

These individual analysis locations gain additional significance when examined in relation to tool maturity. Figure 2.3 visualises the relationship between maturity categories and analysis locations, revealing the following patterns:

C1 Concentration in Onsite - The 11 C1 tools using onsite analysis appear to reflect research prototype characteristics, where researchers prioritise algorithm development over deployment complexity by choosing simple local execution. Early-stage tools have not yet addressed the engineering challenges of cloud deployment or the business model questions of service delivery.

Higher Maturity Diversification - C3, C4, and C5 categories show

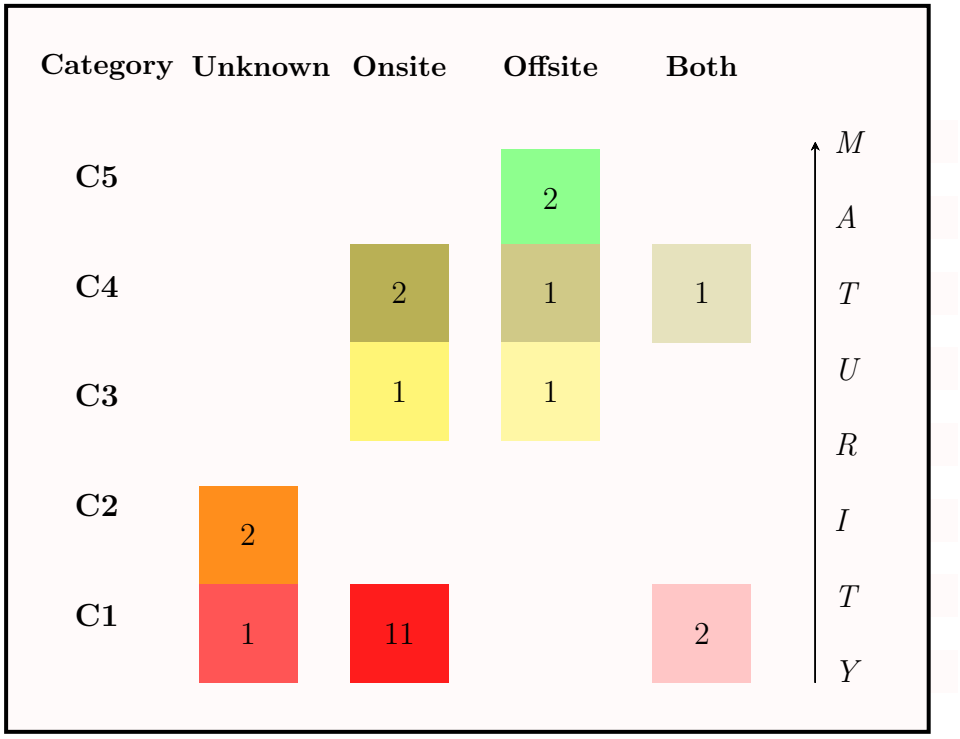


Figure 2.3: Distribution of Static Analysis Tool Maturity Categories by Script Analysis Location

increased location diversity compared to C1. C4 tools distribute across onsite, offsite, and flexible deployment options, while C3 and C5 tools split between onsite and offsite analysis. This evolution reflects production requirements where organisations demand deployment choice. Cloud-based analysis becomes viable as tools mature and establish trust with users.

Proprietary Tool Opacity - The concentration of unknown locations in C2 (proprietary without DevOps integration) suggests these tools may perform offsite analysis but do not document this clearly. This opacity could reflect competitive concerns or security-through-obscurity approaches, but it complicates adoption decisions.

Evolution Toward Flexibility - The presence of flexible deployment options in C1 and C4 indicates that some tools provide deployment choice. This flexibility may represent engineering maturity, as supporting multiple deployment modes requires abstracted architectures and careful security design.

The analysis of location patterns has several implications for tool adoption and development. Organisations with strict security requirements may be lim-

ited to onsite tools, reducing their options primarily to C1 category tools that may lack production features. Tool developers targeting enterprise adoption should prioritise flexible deployment options. The trend toward cloud-based analysis in mature tools suggests this will become the standard, requiring careful attention to security and privacy concerns.

These patterns also suggest a choice between analysis sophistication and deployment constraints. Cloud-based analysis enables more complex techniques through scalable computing resources and centralised intelligence gathering from multiple organisation's patterns. However, security concerns limit adoption, creating opportunity for privacy-preserving analysis techniques that enable cloud benefits without exposing sensitive configurations.

2.7 Discussion

This section presents an overview of current IaC static analysis as identified by the results of this study. It first presents the findings in relation to existing approaches and then discusses how these findings inform the remainder of this thesis.

2.7.1 Observations

From analysing 24 static analysis tools, it can be concluded that the majority of current practices focus on detecting defects rather than providing a comprehensive assessment of IaC scripts. These tools employ various methods, such as regular expression matching, machine learning models, and formal verification, to identify and report specific types of defects within IaC scripts, including anti-patterns, code smells, and security smells. While this defect-focused approach addresses a critical need, it provides an incomplete representation of IaC scripts.

Each of the analysed IaC tools defines its own categories for detecting defects independently of one another. For example, Rahman et al. [8] defined

seven categories of security smells, while Sharma et al. [22] proposed a separate set of classifications. Industry-based tools such as Checkov and KICS each maintain their own rule sets, which are proprietary and do not align with established software engineering standards. As a result, comparing the quality of assessments across tools is challenging.

Additionally, analysed tools adopt a binary approach to assessing IaC scripts, either identifying a defect or not. However, a script that contains no identified defects is not necessarily well-maintained, clearly structured, or configured optimally. There is a broad range of characteristics beyond defect identification that contribute to the maintainability and security of IaC scripts, yet none of the examined tools provide a comprehensive assessment model capable of evaluating and reporting on these additional characteristics. This binary orientation means practitioners selecting between alternative scripts that achieve identical infrastructure objectives have no quantitative criteria for evaluation.

Beyond detection limitations, the examined tools also demonstrate metadata blindness. Each tool analyses scripts individually rather than evaluating collections of scripts at the repository level. While individual script evaluation allows for the collection of metadata on each script, it ignores relevant contextual data that exists among related scripts, such as shared variables, role dependencies, and repository structure. Version history, dependency relationships, and maintenance patterns within a repository provide additional evidence of potential vulnerabilities that remains unused by individual script-based tools. Evaluating a repository's scripts collectively could expose vulnerabilities that are not visible when the same scripts are assessed in isolation.

Finally, the examined IaC tools are unable to analyse relationships between repositories or compare repositories to one another. As a result, practitioners cannot determine whether duplicate code exists across repositories or whether a particular repository's scripts were derived from a common template. Security smells can propagate through code reuse from an original repository to

derived repositories without detection. A capability to analyse and compare repositories would therefore be beneficial for identifying and mitigating the spread of security smells throughout an ecosystem.

2.7.2 Research Direction

These observations reveal fundamental gaps affecting both practitioners and researchers. 14 tools are currently classified at C1 with no DevOps integration. Additionally, there is a disproportionate focus on platform-specific research (e.g., analysing only Ansible scripts, or only Terraform scripts) relative to cross-platform comparisons. Much of the existing research does not appear to be transitioning into practice, resulting in a widening gap between academic contributions and practitioner needs. To address these gaps, a necessary first step is the development of a standardised framework grounded in well-established software engineering concepts that moves beyond defect detection towards transparent and systematic measurement. Such a framework would identify common characteristics across all IaC technologies, define measurable metrics for each attribute, and enable comparative quantification between alternative implementations. Using this as a foundation, the framework could also address the other observations identified in this analysis, including the incorporation of metadata and repository-level analysis, as well as comparative analysis between repositories. The next chapter presents this framework, which establishes a methodology for comprehensive assessment of IaC that bridges the gap between research innovation and practitioner needs. Table 2.4 makes this mapping explicit, relating each gap identified in the review to the research contributions defined in Chapter 1 and the chapters that address them, complementing the challenge mapping presented in Figure 1.1.

The review does not extend to AI-assisted code generation, which emerged as a practical concern only towards the end of the surveyed period. The fourth contribution (RC4), the quality assessment of AI-generated IaC scripts in Chapter 7, is motivated instead by the deployment-velocity challenge de-

Table 2.4: Mapping of the gaps identified in the review to research contributions and thesis chapters.

Gap Identified in the Review	Research Contribution	Chapter
Current tools detect specific defects but provide no comprehensive assessment of IaC scripts	RC1: IaC quality framework and policy-driven quality assessment	3, 4
Detection categories are defined independently per tool and do not align with established software engineering standards	RC1: IaC quality framework and policy-driven quality assessment	3
Binary defect reporting provides no quantitative criteria for comparing alternative implementations	RC1: IaC quality framework and policy-driven quality assessment	4
Script-level analysis ignores repository metadata such as version history, dependency relationships, and maintenance patterns	RC2: Metadata-assisted supply-chain attack detection	5
Relationships between repositories are not analysed, allowing security smells to propagate through code reuse undetected	RC3: Security smell diffusion analysis	6

scribed in Chapter 1, and it builds directly on the comparative capability that closes the third gap above.

2.8 Chapter Summary

This chapter analysed 24 tools for static configuration analysis in IaC scripts, revealing that regular expressions remain the most common detection technique , used by 13 of the 24 tools despite limitations in automation and adaptability, while machine learning and formal methods show promise but face adoption barriers including data scarcity and complexity.

Our analysis also identified gaps in tool coverage, particularly for configuration management and image building categories, with most tools remaining in early maturity stages without integration with production environments.

The field of IaC has experienced tool growth from 2011 to 2023, with notable acceleration during the COVID-19 pandemic driven by increased demand for infrastructure automation. Despite this growth, the progression of tools from research prototypes to production-ready solutions remains slow. Our analysis revealed three limitations. First, each tool defines its detection cat-

egories independently, without reference to established software engineering standards. Second, current tools operate on a binary basis, identifying defects without providing holistic assessment capabilities. Third, no mechanism exists for comparing alternative implementations that achieve the same infrastructure objectives.

These observations suggest that current defect-oriented approaches, while valuable, may not address the broader need for systematic IaC assessment. The next chapter addresses the issue by presenting a framework for comprehensive IaC assessment grounded in recognised software engineering principles, establishing measurable characteristics applicable across IaC technologies and enabling quantitative comparison between alternative implementations.

Chapter 3

An IaC Quality Framework

Statistics presented in Chapter 1 documents the operational and financial consequences of infrastructure misconfigurations. This chapter proposes an IaC quality framework to address these issues by enabling systematic identification of problematic patterns before deployment, comparison between alternative implementations, and tracking of improvements over time.

The preceding chapter revealed that current IaC tools cannot fulfil these functions. These tools have developed in a piecemeal fashion, with each defining its own detection categories independently and operating in binary terms. They lack reference to established software engineering standards and exhibit metadata blindness by processing scripts individually without incorporating repository-level characteristics. This fragmentation makes it difficult to compare infrastructure scripts across platforms, establish organisational quality baselines, and track quality changes over time. What is lacking is a unified approach to IaC quality assessment grounded in established software engineering principles yet tailored to infrastructure-specific concerns. The benefits of such a framework are manifold: transparency in assessment methodology and terminology, structured decomposition of quality into measurable dimensions, and consistent evaluation through empirically grounded measurement approaches.

This chapter presents a quality framework for systematic IaC assessment. At its core, the framework extends ISO/IEC 25010 principles to accommodate

infrastructure-specific concerns through nine categories derived from analysis of IaC characteristics. These categories move beyond binary smell detection to capture attributes that indicate overall script quality, including metadata indicators that reflect repository maturity and community adoption. Within each category, each attribute is extracted, normalised across repositories, weighted according to policy configurations, and aggregated to produce quantitative measurements. To develop this framework, the chapter begins by examining quality evaluation mechanisms across IaC platforms to establish the current landscape. A review of prior research follows, identifying quality attributes proposed for specific IaC platforms. The chapter then establishes the rationale for standards-based assessment before analysing where ISO/IEC 25010 accommodates infrastructure code and where limitations emerge. These identified gaps inform the proposed categories and algorithm.

3.1 Related Work

This section examines existing IaC quality assessment practices and research to establish the context for the proposed framework. The analysis builds upon three IaC categories discussed in Section 2.1.1, namely Infrastructure Provisioning, Configuration Management, and Image Building. Each category employs distinct platforms and codebases with varying evaluation mechanisms, necessitating examination of their respective assessment approaches.

3.1.1 Quality Assessment Mechanisms Across IaC Platforms

IaC platforms employ varying mechanisms to evaluate script quality across codebases, with substantial heterogeneity in assessment approaches. Table 3.1 documents the quality metrics implemented by each platform category. Configuration management platforms such as Ansible Galaxy, Puppet Forge, and Chef Supermarket provide quantitative scoring systems with scales such as 5

point Likert scales and percentage-based metrics. Infrastructure provisioning platforms largely lack quality assessment mechanisms, with Azure Resource Manager Templates providing only binary Pass/Fail evaluation. Image building tools either lack formalised metrics or employ verification programmes rather than quantitative scores. This inconsistency complicates cross-platform quality comparison and baseline establishment.

Table 3.1: Overview of IaC Platforms and Associated Quality Metrics by Category

IaC Categories	IaC Platforms	IaC Codebases	Quality Scoring Metrics
Infrastructure Provisioning	Terraform [45]	Terraform Registry [46]	–
	CloudFormation (AWS)	–	–
	Azure Resource Manager (ARM) Templates [47]	Azure Quickstart Templates [48]	Pass/Fail
	Google Cloud Deployment Manager [49]	–	–
Configuration Management	Ansible [18]	Ansible Galaxy	Scale of 0-5 points
	Puppet [50]	Puppet Forge [51]	Scale of 0-100%
	Chef [52]	Chef Supermarket	Scale of 0-100%
	SaltStack (Salt) [53]	SaltStack Formula [54]	–
Image Building	Packer	–	–
	Docker	Docker	Verified Publisher Program

Beyond the presence or absence of quality scoring, the implemented systems themselves exhibit a wide variety of approaches. Table 3.2 examines these differences across platforms that provide quality metrics.

Table 3.2: Comparison of Quality Scoring Systems across IaC Platforms

Feature	Ansible Galaxy	Puppet Forge	Chef Supermarket	Azure	Docker
Method	Community & quality scores using user surveys (deprecated)	Automated code/malware checks	Community metrics & platform support	Compliance checks	Security scanning & verification
Scoring Components	Community, syntax	Formatting, security	Publication, licensing, support	Compliance, security with Azure best practices	Vulnerabilities, integrity, publisher verification
Tools	yamllint, ansible-lint	VirusTotal	Foodcritic	Azure Policy, ARM Toolkit	Docker Scout
Limitations	Survey dependence	Limited docs	Community feedback reliance	Low transparency	Low transparency

Ansible Galaxy implemented a community-driven quality score ranging from 0 to 5 points. The platform utilised user surveys to assess repository qual-

ity, incorporating syntax validation through `yamllint` [1] and `ansible-lint` [2] open-source tools. The survey-based approach was deprecated and applied to only 497 repositories, limiting its coverage across the platform. [3] The scoring components focused on community engagement and syntax correctness, but dependence on user participation constrained assessment consistency.

Puppet Forge integrates *VirusTotal* for security scanning and malware detection, producing automated quality scores from 0 to 100%. The platform evaluates code formatting and security vulnerabilities but not code quality in broader terms. Documentation on score calculation remains sparse, reducing transparency in how practitioners should interpret results.

Chef Supermarket also utilises a percentage-based system (0-100%) combining community metrics with platform support indicators. The platform employs *Foodcritic* for code analysis, license check, and platform compatibility. The system relies on community feedback for quality signals, introducing variability in assessment reliability.

Azure Quickstart Templates implement a binary Pass/Fail system through compliance checks. The platform uses Azure Policy and ARM Toolkit to verify adherence to Azure best practices and security standards. Although this binary approach provides unambiguous assessment results, it lacks granularity for differentiating quality levels among repositories that pass.

Docker Hub takes a different approach entirely, operating a Verified Publisher Program without numerical scoring. Docker Scout performs vulnerability scanning and integrity verification, but the absence of quantitative metrics means repositories cannot be compared systematically.

Such variation in scoring methods, ranging from community-driven ratings to automated compliance checks, impedes cross-platform quality comparison. The analysis reveals two additional concerns. First, transparency remains

[1] Adrien Cunin, "yamllint Documentation: A Linter for YAML Files," 2023, https://yamllint.readthedocs.io.

[2] Will Thames and contributors, "ansible-lint: Ansible Playbook Linter," 2023, https://ansible-lint.readthedocs.io.

[3] Ansible Galaxy Content Scoring Documentation, https://old-galaxy.ansible.com/docs/contributing/content_scoring.html.

inconsistent as platforms such as Puppet Forge provide limited documentation on score calculation while Docker Hub lacks quantitative metrics entirely. The transparency problem mirrors findings from the preceding chapter, where IaC analysis tools similarly lack documentation on detection methodologies and category definitions. Second, no platform offers comprehensive assessment spanning multiple quality dimensions, with each focusing on isolated concerns such as security scanning, compliance verification, or community engagement. These inconsistencies complicate infrastructure script selection and quality baseline establishment.

3.1.2 Prior Research

Prior research acknowledges this limited consensus while proposing various quality assessment approaches. As detailed in this section, research initiatives range from tool-specific detection mechanisms to quality metric establishment across individual IaC platforms.

In providing empirical quality indicators, Rahman et al. [4] investigated source code properties correlated with defects in Puppet scripts. Defects in this context include configuration errors, incorrect resource specifications, missing dependencies, and security misconfigurations that cause infrastructure provisioning failures or unexpected system states. Their study analysed 262 Puppet repositories containing 8,268 scripts from organisations including Mozilla, OpenStack, and Wikimedia. Through statistical analysis, the researchers identified 12 source code properties exhibiting correlation with defect occurrence, including lines of code (correlation coefficient: 0.43), hard-coded strings (0.38), and file mode specifications (0.41). The defect prediction models constructed using these properties achieved precision values ranging from 0.70 to 0.78 and recall values from 0.54 to 0.67 across different organisational datasets. The properties identified include statements, ensure attributes, and SSH key specifications which demonstrate tool-specific characteristics that may not translate to other IaC platforms employing different syntax and paradigms.

Addressing quality assessment for configuration management tools beyond Puppet, Dalla Palma et al. [5] developed a catalogue of quality attributes for Ansible scripts. Their methodology involved analysing 100 Ansible Galaxy repositories and conducting interviews with 14 practitioners to identify quality concerns. The resulting catalogue comprises 46 code-level attributes. Attributes such as average play size, entropy measurements, and use of deprecated modules provide metrics for assessment. The researchers implemented static analysis tools to extract these attributes, demonstrating feasibility of automated measurement. However, the catalogue lacks empirical validation linking attributes to actual quality outcomes, and the absence of weighted importance among attributes prevents prioritisation.

Complementing configuration management research, Begoug et al. [55] introduced TerraMetrics, a tool measuring quality in Terraform scripts. The tool implements 40 code-level attributes derived from static code analysis, incorporating metrics similar to those identified by Dalla Palma et al. while adapting to Terraform's declarative syntax. TerraMetrics evaluates aspects including resource dependencies, module usage patterns, and variable declarations specific to infrastructure provisioning. Validation on three Terraform repositories demonstrated the tool's capability to identify quality variations, with scores ranging from 42% to 78% across repositories. The limited validation scope and absence of correlation with actual infrastructure failures or maintenance efforts constrains confidence in the tool's effectiveness. The metrics focus on syntactic properties without addressing semantic correctness or infrastructure design quality.

Beyond individual platform studies, Section 2.3.2 surveyed 24 IaC quality detection tools and identified significant fragmentation in existing methodologies. Current tools operate separately, targeting individual issue categories without delivering integrated quality analysis. The survey found that 73% of tools concentrate on code and security vulnerabilities, while 42% target maintainability issues, yet no tool provides comprehensive evaluation spanning mul-

tiple quality aspects. Such fragmentation highlights the lack of standardised approaches to IaC quality assessment.

Synthesis of these studies reveals patterns in IaC quality research. Each investigation targets specific platforms such as Puppet, Ansible, or Terraform, resulting in tool-dependent quality models. The collective identification of over 100 distinct quality attributes across studies indicates the complexity of IaC quality assessment, yet overlap remains minimal due to platform differences. Rahman et al.'s statistical approach provides empirical grounding but lacks the breadth of Dalla Palma et al.'s practitioner-informed catalogue. TerraMetrics demonstrates automated assessment feasibility for infrastructure provisioning tools, contrasting with the configuration management focus of other studies. Beyond platform fragmentation, all examined studies focus solely on code-level properties while excluding metadata attributes such as version history, update frequency, community metrics, and dependency information that indicate script maturity and maintenance patterns. This fragmentation across IaC quality research contrasts with traditional software engineering, where quality standards have provided unified assessment frameworks for decades.

3.1.3 The Role of Standards in Software Quality Assessment

Understanding how standards contribute to quality assessment in traditional software contexts informs the development of corresponding frameworks for IaC. Software quality is inherently multidimensional, and widely used quality models decompose it into multiple characteristics (e.g., reliability, security, maintainability) that together determine a product's fitness for purpose [56]. However, these characteristics are abstract, and a recurring challenge is how to operationalise them into concrete, valid measures and assessments in practice [56, 57]. Standards and standardised quality models (e.g., ISO/IEC 9126:1991 and its successors) address this by providing agreed terminology and a structured decomposition of quality into characteristics and sub-characteristics that

can be specified, measured, and evaluated across the lifecycle [58]. Such standards are typically produced via consensus-based processes which helps externalise expert knowledge into repeatable frameworks for communication and evaluation.

The relationship between standards adherence and quality outcomes operates through several mechanisms. Standards and quality models can encode accumulated lessons as recommended practices and checklists as well as enable measurement by mapping abstract quality concepts to observable factors. They also facilitate comparison by providing more consistent evaluation criteria across projects and organisations [57]. Empirical studies by Stamelos et al. [59] have demonstrated the value of adapting standards to specific domains. The authors in [59] assessed 100 Linux applications using criteria derived from ISO/IEC 9126 [58] and found that structural code measurements correlated with user satisfaction. This correlation supported the adoption of coding standards and automated analysis for improving release quality. Overall, standards-based frameworks provide a defensible foundation for quality assessment when they are tailored to the target domain and validated for the intended usage scenario [57, 60].

3.1.4 Software Quality Characteristics in ISO/IEC 25010

ISO/IEC 25010:2011 [12] exemplifies this standards-based approach, defining software product quality as part of the Systems and Software Quality Requirements and Evaluation (SQuaRE) series. The standard supersedes ISO/IEC 9126 [14], introducing refined quality characteristics and expanding coverage of quality concerns. The framework aims to provide common terminology and understanding for specifying, measuring, and evaluating software quality across stakeholders including developers, acquirers, and quality assurance professionals.

The standard comprises two complementary models addressing different quality perspectives. The Quality in Use Model defines five characteristics

which are effectiveness, efficiency, satisfaction, freedom from risk, and context coverage thus focusing on outcomes achieved when software is used in specific contexts. The Product Quality Model specifies eight characteristics examining inherent properties of software products: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. Each characteristic decomposes into sub-characteristics providing granular quality aspects.

Application of ISO/IEC 25010:2011 to IaC requires consideration of the Product Quality Model rather than the Quality in Use model. The Product Quality Model evaluates intrinsic properties of software artifacts, whereas the Quality in Use model assesses quality from the perspective of users interacting with a system in a specific context. IaC scripts constitute software artifacts with inherent properties such as structure, syntax, and configuration patterns that can be evaluated independently of execution context. We map all eight characteristics to various aspects of IaC usage for configuration management:

- **Functional suitability** addresses whether scripts provision correct resources.
- **Performance efficiency** concerns execution time and resource consumption.
- **Compatibility** examines integration with platforms and tools.
- **Usability** relates to script readability and documentation.
- **Reliability** encompasses error handling and idempotency.
- **Security** covers credential management and access controls.
- **Maintainability** includes modularity and reusability.
- **Portability** addresses cross-platform execution.

However, the standard exhibits limitations when applied to IaC contexts. The characteristics and sub-characteristics remain abstract, providing defini-

tions without concrete metrics or measurement methods. For example, the Maintainability characteristic includes a sub-characteristic "Modularity" defined as *"the degree to which a system is composed of discrete components such that a change to one component has minimal impact on other components"* [12]. This definition does not specify what constitutes a component in IaC contexts, whether a task, role, or playbook, nor does it quantify what "minimal impact" means. Similarly, the Reliability sub-characteristic "Fault Tolerance" is defined as *"the degree to which a system operates as intended despite the presence of hardware or software faults,"* [12] yet provides no guidance on measuring fault tolerance in infrastructure scripts that configure external systems. The standard thus provides a taxonomy for categorising quality concerns rather than an operational framework for assessment. Organisations applying the standard to infrastructure code would need to independently define measurement approaches for each characteristic, leading to inconsistent interpretations across teams and projects.

The model's development predates the widespread adoption of infrastructure automation. IaC introduces concerns that the standard does not address, including infrastructure state management, declarative versus imperative paradigms, idempotency requirements, configuration drift, and repository metadata. The standard's sub-characteristics do not encompass these infrastructure-specific aspects, as no sub-characteristic addresses configuration convergence, resource dependency management, or metadata analysis unique to IaC.

Furthermore, ISO/IEC 25010:2011 assumes traditional software development and deployment models. The characteristics focus on software as discrete applications rather than infrastructure definitions that instantiate infrastructure resources. As a result, quality concerns specific to IaC fall outside the standard's scope, including compliance with infrastructure providers, multi-region deployment strategies, and the dual nature of IaC as both code and infrastructure specification.

These limitations necessitate extending ISO/IEC 25010 principles while developing IaC-specific quality categories. The standard provides foundational concepts and establishes quality terminology, however lacks the specificity to be applicable to infrastructure automation in its current form. Existing IaC quality research has not bridged this gap, as none of the studies discussed in Section 3.1.2 align their attributes with ISO/IEC 25010 or similar standards. To identify where the standard needs to be augmented for this specificity, this work conducts a mapping exercise that systematically examines where ISO characteristics correspond to IaC attributes and where extensions need to be made. This analysis informs the development of a framework that maintains alignment with international standards while addressing the unique quality requirements of infrastructure as code.

3.2 Deriving Quality Metrics from IaC Code Attributes

Section 2.1.2 discussed the code-level attributes and repository metadata that are found in IaC scripts. Both categories of attributes are important indicators of IaC quality. Code-level attributes represent the structural elements and correctness of a script, while repository metadata represents community validation, maintenance history, and use patterns. In this section, we will be applying these categories of attributes to ISO/IEC 25010 quality characteristics to create an IaC quality framework. Our approach aims to address a gap in existing research that has focused solely on code-level characteristics while ignoring the quality indicators associated with metadata.

Building on this foundation, we examine the relationship between IaC attributes and quality frameworks through the following three steps:

- *Step 1: Mapping IaC Code Attributes to ISO/IEC 25010:2011.* This mapping examines the extent to which the ISO/IEC 25010:2011 standard accommodates IaC code attributes mentioned in related works of

Rahman et al. [4] & Dalla Palma et al. [5].

- *Step 2: Development of IaC Code Quality Framework.* This step develops framework categories that address IaC requirements where ISO/IEC 25010:2011 has limited applicability. The derivation process for this step is described in Section 3.2.2.

- *Step 3: Mapping IaC Code Attributes to IaC Code Quality Framework.* This step presents comparative analysis, demonstrating the framework's capacity to capture IaC-specific characteristics compared with ISO/IEC 25010:2011 standard.

3.2.1 Mapping IaC Code Attributes to ISO/IEC 25010:2011

The mapping of existing IaC code quality attributes to ISO/IEC 25010:2011 characteristics examines the extent to which the standard accommodates infrastructure code assessment. This exercise evaluates attributes identified in prior research against the eight characteristics of the Product Quality Model to determine alignment and identify gaps. The mapping categorises attributes into three types: *Direct Mapping* where IaC properties correspond to ISO characteristics by definition, *Indirect Mapping* where attributes align operationally without direct correspondence, and *No Mapping* where attributes remain unique to IaC without fitting the ISO framework as their definition are not aligned. The work by Begoug et al. [55] was excluded from mapping given the close alignment with Dalla Palma et al. [5] in attribute identification methodology, avoiding redundant analysis.

3.2.1.1 Mapping Rahman et al.'s Puppet Attributes to ISO/IEC 25010:2011

Rahman et al. [4] identified 12 source code properties correlated to identify defects in Puppet scripts through empirical analysis. We mapped these properties to ISO/IEC 25010:2011, which reveals incomplete coverage by the ISO

standard. Table 3.3 presents our complete mapping results.

Table 3.3: Mapping of IaC Code Attributes from Rahman et al.'s [4] to ISO/IEC 25010:2011 Quality Characteristics

ISO 25010:2011 Characteristics	Direct Mapping	Indirect Mapping
Functional Suitability	Include, Ensure, Require, Attribute	✕
Performance Efficiency	✕	✕
Compatibility	✕	✕
Usability	Comment	✕
Reliability	Ensure, File	✕
Security	SSH Key, File Mode	Hard-Coded String
Maintainability	File	✕
Portability	✕	Lines of Code

No Mapping
URL, Command

Legend:
✕ - No IaC code attributes maps to ISO 25010:2011 characteristics

The mapping exercise identifies correspondences for attributes that align with ISO characteristics. The "Include" and "Require" statements map to Functional Suitability as they specify required dependencies for proper script execution. "Ensure" attributes map to both Functional Suitability and Reliability, defining desired system states. "File" specifications relate to Reliability and Maintainability through resource management. "SSH Key" and "File Mode" attributes directly map to Security characteristics. The "Comment" attribute maps to Usability, supporting code comprehension.

The analysis also identifies indirect mappings where attributes impact quality without explicit ISO representation. "Hard-Coded String" indirectly relates to Security, as embedded credentials or configurations pose security risks. "Lines of Code" indirectly affects Portability, as larger scripts exhibit reduced transferability across environments. These attributes influence quality characteristics despite lacking direct sub-characteristic correspondence definitions.

Attributes with no mapping highlight gaps in the ISO standard for IaC assessment. The "URL" metric, tracking external resource references, and "Command" executions lack representation in the standard. These infrastructure-specific properties demonstrate limitations in traditional software quality models when applied to IaC contexts.

3.2.1.2 Mapping Dalla Palma et al.'s Ansible Attributes to ISO/IEC 25010:2011

Dalla Palma et al. [5] expanded quality assessment to 46 attributes for Ansible scripts, providing broader coverage of IaC characteristics. The mapping exercise reveals alignment of IaC-specific properties with ISO standards, as shown in Table 3.4.

Table 3.4: Mapping of Code Quality Attributes from Dalla Palma et al.'s [5] to ISO/IEC 25010:2011 Quality Characteristics

ISO 25010:2011 Characteristics	Direct Mapping	Indirect Mapping
Functional Suitability	Include, Ensure	Variables
Performance Efficiency	Loops	Regex, Conditions, Decisions, Math Operation
Compatibility	×	×
Usability	×	User Interactions
Reliability	Ensure, File Mode, File, Error Handling, Ignore Errors	Blocks
Security	SSH Key, File Mode	Suspicious Comments, Deprecated Keywords, Deprecated Modules
Maintainability	Comment	Line Blank, Line Source Code, Blank Space Between Words, Avg Play Size, Avg Task Size, Unique Names, Length of Tasks, Entropy, Blocks, Variable Names
Portability	Paths, Import Playbook, Import Role, Include Role, Include Task, Import Task, Include Vars	Blocks

No Mapping
Play, Roles, URL, Commands, Fact Modules, Filters, Lookups, Keys, (External, Distinct) Modules

Legend:

× - No IaC code attributes maps to ISO 25010:2011 characteristics

Direct mappings encompass attributes aligning with quality characteristics. Loop constructs map to Performance Efficiency through execution optimization concerns. Error handling mechanisms including "Ignore Errors" map to Reliability. Import and include statements for playbooks, roles, tasks, and variables map to Portability, enabling code reuse across environments. Security mappings include "SSH Key" and "File Mode" specifications, consistent with Rahman et al.'s [4] findings.

Indirect mappings demonstrate operational quality impacts without direct ISO correspondence. "Variables" usage indirectly supports Functional Suitability through parametrisation. Computational constructs including "Regex," "Conditions," "Decisions," and "Math Operations" relate to Performance Efficiency through processing complexity. "User Interactions" indirectly affect Usability. Code organisation metrics such as "Line Blank," "Avg Play Size," and "Entropy" influence Maintainability without explicit sub-characteristic alignment. "Suspicious Comments" and "Deprecated Keywords/Modules" present indirect Security concerns through potential vulnerabilities.

The substantial set of unmapped attributes underscores the ISO standard's limitations. Infrastructure-specific constructs including "Plays," "Roles," "Filters," "Lookups," "Fact Modules," and "External Modules" lack ISO representation. These elements constitute fundamental IaC components yet remain outside traditional quality frameworks. The presence of "URL" and "Commands" attributes, similar to Rahman et al.'s [4] findings, confirms persistent gaps across IaC platforms.

The mapping results from both studies reveal increasing divergence between IaC code attributes and ISO characteristics as assessment scope expands. We find that, while core software properties achieve mapping, infrastructure-specific elements consistently lack representation. This indicates the need for specialised quality frameworks accommodating IaC characteristics.

3.2.2 Framework Derivation Methodology

To address the gaps exposed by the mapping, we derived nine IaC quality categories through three activities. We examined ISO/IEC 25010:2011, published IaC literature, and official platform documentation, bringing together the definitions in the standard, empirically studied attributes, and common quality elements across platforms. Figure 3.1 shows how these activities contributed to the category definitions, which Section 3.2.3 presents.

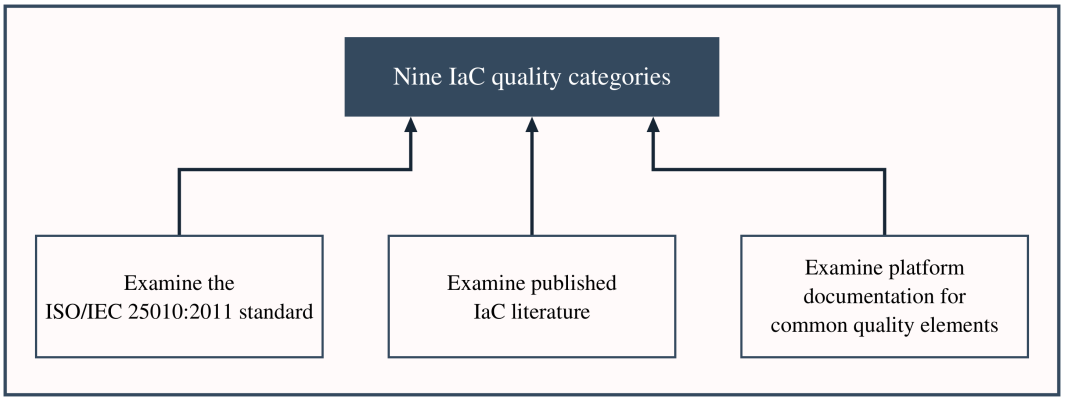


Figure 3.1: Three activities informing the derivation of the nine IaC quality categories.

Examination of ISO/IEC 25010:2011

We examined each characteristic, its sub-characteristics, and their definitions in ISO/IEC 25010:2011 [12]. We used these definitions rather than characteristic names alone because the mapping depended on what each characteristic covered. This examination provided the basis for assessing whether the standard could express the quality concerns reported in IaC research.

Examination of published IaC literature

We examined the twelve defect-correlated Puppet properties identified by Rahman et al. [4] and the forty-six Ansible attributes catalogued by Dalla Palma et al. [5]. Rahman et al. derived their properties from defect analysis, while Dalla Palma et al. combined repository analysis with practitioner interviews. These catalogues provided empirical inputs from two IaC platforms.

For each attribute, we compared its published definition with the ISO characteristics and sub-characteristics, recording a direct, indirect, or no-mapping classification. Where the relationship concerned an operational effect without explicit correspondence to an ISO definition, we recorded an indirect mapping rather than a direct one because the published definitions did not establish a direct correspondence. The classifications presented in the preceding subsection identify the concerns that the standard accommodates and those requiring further interpretation or extension for IaC.

Examination of platform documentation

We examined official documentation to identify how quality concerns are expressed through platform mechanisms. We selected platforms using the codebase criterion recorded in Table 3.1, rather than familiarity, because publicly accessible collections of IaC artefacts provide the setting in which practitioners encounter repository and platform-level quality signals. The selected platforms were Terraform, Azure Resource Manager, Ansible, Chef, Puppet, SaltStack, and Docker. Table 3.1 does not record a qualifying codebase for CloudFormation, Google Cloud Deployment Manager, or Packer, so these platforms were excluded from the documentation comparison. Puppet contributed both a published attribute catalogue and platform documentation.

We examined mechanisms for repository metadata, code structure, language features, documentation and readability, secret handling, syntax validation and licensing, error management, automation, and component composition. The documentation sources are listed in the accompanying foot-

notes.[4] [5] [6] [7] [8] [9] [10]

Tables 3.5 and 3.6 record the relationship between each category and the documented platform mechanisms. Following the grouping in Section 2.1.1, the first table covers infrastructure provisioning and image building, and the second covers configuration management. The table entries were checked against the documentation on the access dates recorded in the footnotes.

For an element to contribute to a shared category, we required a documented mechanism across all seven platforms. Platform-specific elements, including Terraform state backends, Ansible inventories, Chef node attributes, Salt pillar targeting, and Docker layer caching, were not elevated to separate categories because they did not meet this cross-platform criterion.

We brought these inputs together by reinterpreting the ISO characteristics in the context of the published attributes and documented platform mechanisms. Where an ISO characteristic accommodated an infrastructure concern, the category retained that relationship, as recorded in the sub-characteristic lists accompanying its definition. Where the mapping exposed a concern that the standard did not express, we defined a category around that concern. This process produced the nine categories defined in the following subsection. The platform examples show how the categories apply across the selected platforms. They do not establish that nine is the only possible categorisation.

The derivation used published attribute catalogues and platform documentation rather than large-scale mining of scripts across all seven platforms

[4] HashiCorp, "Terraform Documentation," https://developer.hashicorp.com/terraform/docs, accessed 9 September 2026.

[5] Microsoft, "Azure Resource Manager Templates Documentation," https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/, accessed 16 September 2026.

[6] Ansible Project, "Ansible Documentation," https://docs.ansible.com, accessed 9 September 2026.

[7] Progress Chef, "Chef Infra Client 19 Documentation," https://docs.chef.io, accessed 9 September 2026.

[8] Perforce Puppet, "Puppet Core Documentation," https://help.puppet.com/core/current/, accessed 16 September 2026.

[9] Salt Project, "Salt Documentation," https://docs.saltproject.io/en/latest/, accessed 16 September 2026.

[10] Docker Inc., "Docker Documentation," https://docs.docker.com, accessed 9 September 2026.

Table 3.5: Common Quality Elements Across IaC Platform Documentation: Infrastructure Provisioning and Image Building

IaC Quality Category	Infrastructure Provisioning		Image Building
	Terraform	**Azure Resource Manager**	**Docker**
Metadata	Registry standard module structure and generated module documentation	Template `metadata` object, parameter `metadata.description`	`LABEL` image metadata, including the `org.opencontainers.image` annotation keys
Code Structure	Standard module layout (`main.tf`, `variables.tf`, `outputs.tf`)	Linked and nested templates	Ordered instruction format beginning with `FROM`, `.dockerignore`
Code Sophistication	Expressions, functions, dynamic blocks	Template functions, `copy` loops, `condition` element	Multi-stage builds, `ARG` build arguments
Code Maintainability	README per module, variable and output descriptions	Description for every parameter in `metadata`	Comments, `LABEL` for image organisation
Code Security	`sensitive` variables	`secureString` parameters, Key Vault references	Build secret mounts, non-root `USER`
Functionality & Purpose	`terraform validate`, licence file in standard structure	ARM template test toolkit (`arm-ttk`)	Build-time instruction validation (`docker build --check`), `EXPOSE` port documentation
Error Handling	Variable validation, preconditions and postconditions	`dependsOn` ordering, `--rollback-on-error`	`HEALTHCHECK`
Code Automation	Declarative state reconciliation, drift detection	Incremental deployment mode, what-if preview	Automated image builds from declarative instructions
Code Integration	Module composition, nested module calls	Linked template composition	`FROM` base-image composition, `ONBUILD`

because the catalogues supplied empirical attributes and the documentation explained the corresponding platform mechanisms. The later application to repository data, reported in Chapter 4, examines the use of the framework at ecosystem scale. It is separate from the derivation described here.

Using published definitions made the basis for each mapping explicit, but the interpretation of those definitions still depended on our judgement. A borderline attribute could be classified differently by another researcher. We considered evidence from the standard, both catalogues, and the seven documentation sets when defining the categories. As future work, we plan an inter-rater agreement study in which industry practitioners would independently map the attributes to ISO/IEC 25010:2011 and the proposed framework. Comparing

Table 3.6: Common Quality Elements Across IaC Platform Documentation: Configuration Management

IaC Quality Category	Configuration Management			
	Ansible	**Chef**	**Puppet**	**SaltStack**
Metadata	Role metadata in `meta/main.yml`, Galaxy platform metadata	`metadata.rb` required in every cookbook	`metadata.json` required in every module and by the Forge	SPM `FORMULA` file (name, version, summary, description)
Code Structure	Defined role directory structure with seven standard directories	Defined cookbook directory structure	Standard module layout (`manifests/`, `templates/`, `files/`, `init.pp`)	Formula layout (`init.sls`, `map.jinja`, `pillar.example`)
Code Sophistication	Loops, conditionals, filters, lookups	Ruby DSL resources and recipes	Defined resource types, iteration functions (`each`, `map`, `filter`)	Jinja templating, requisites
Code Maintainability	README, comments	README, Cookstyle style cops	README, Puppet Strings generated documentation	Formula README, `pillar.example`
Code Security	Ansible Vault encrypted variables	Encrypted data bags, Chef Vault	`Sensitive` data type, Hiera eyaml encrypted data	Pillar for sensitive data, GPG renderer
Functionality & Purpose	`yamllint`, `ansible-lint` syntax validation	Cookstyle correctness and licence cops, `metadata.rb` licence field	`puppet parser validate`, `license` field in `metadata.json`	`state.show_sls` rendering check
Error Handling	Blocks, rescue, `failed_when`, retries	Exception and report handlers	`fail()` function, `noop` metaparameter	`onfail`, `onchanges` requisites, `failhard`
Code Automation	Idempotent modules, desired-state convergence	Test-and-repair convergence	Desired-state resource convergence on every agent run	Declarative state enforcement, `highstate`, `test=True` dry run
Code Integration	Role dependencies, import and include of roles	`depends`, `include_recipe`	`dependencies` in `metadata.json`, class `include`	`include`, `require` and `watch` requisites

their mappings with ours would assess agreement on attribute placement and category boundaries.

3.2.3 Proposed IaC Code Quality Framework

Based on the derivation in Section 3.2.2, we propose a framework of nine quality categories that extends the principles of ISO/IEC 25010:2011 [12] while addressing the requirements of infrastructure code. Figure 3.2 shows how the categories align with the ISO characteristics.

The nine categories are defined as follows. Each category lists the related ISO/IEC 25010:2011 characteristics with their corresponding sub-characteristics

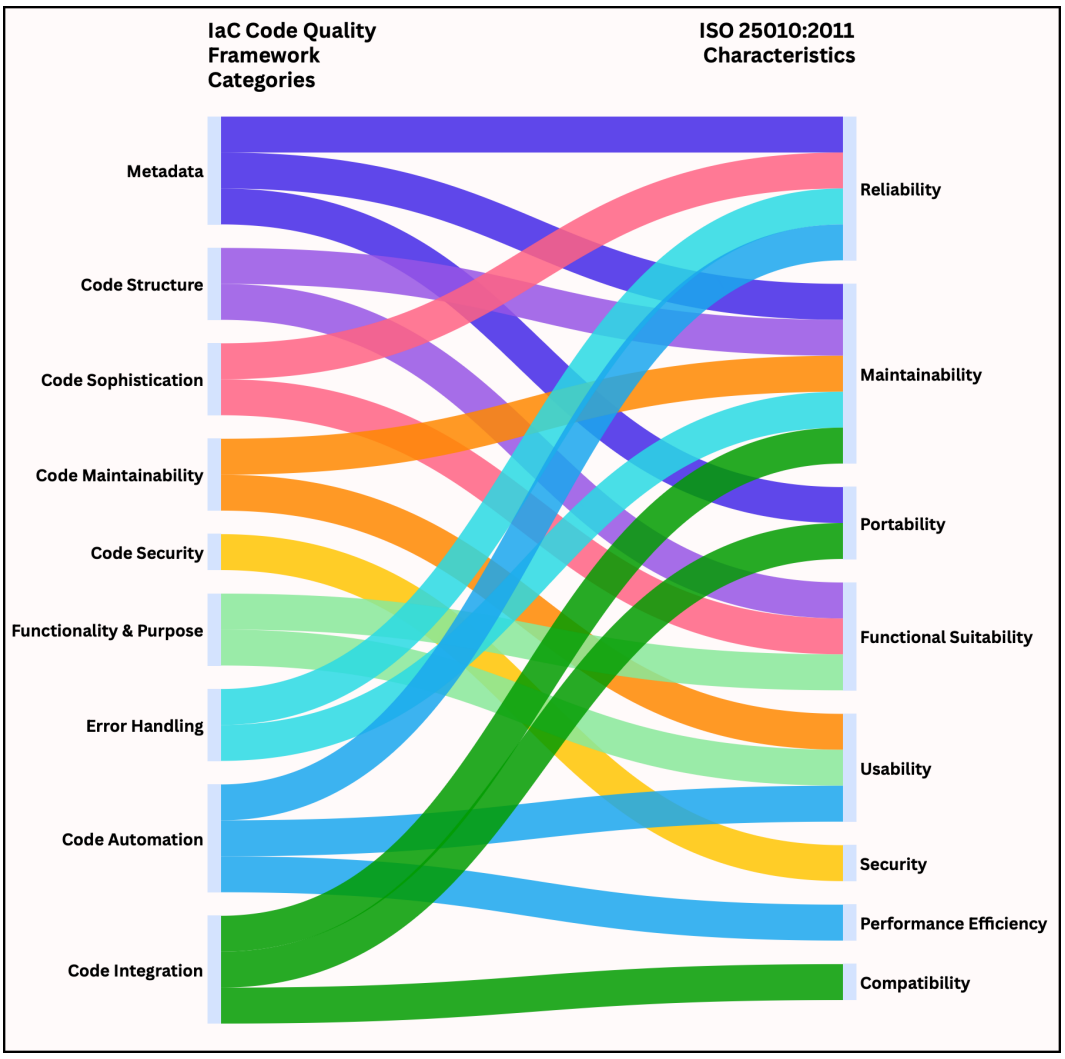


Figure 3.2: Alignment of Proposed IaC Code Quality Framework with ISO/IEC 25010:2011

in parentheses.

1. ***Metadata*** measures repository maturity, maintenance patterns, adoption, and community trust through platform-derived metrics and repository metadata characteristics. The related ISO characteristics are:

- Reliability (Maturity)
- Maintainability (Analysability)
- Portability (Installability, Adaptability)

2. ***Code Structure*** evaluates repository quality and architectural compliance against IaC platform guidelines, measuring adherence to recommended directory structures and component completeness. The related ISO characteristics are:

- Functional Suitability (Functional Completeness)
- Maintainability (Modularity, Testability, Analysability)

3. ***Code Sophistication*** assesses technical maturity of a repository through utilisation of advanced platform features and the ability to handle complex scenarios using various code attributes. The related ISO characteristics are:

- Reliability (Maturity)
- Functional Suitability (Functional Appropriateness, Functional Completeness)

4. ***Code Maintainability*** measures ease of understanding, modifying, and sustaining scripts over time through documentation quality, code readability, and interaction mechanisms. The related ISO characteristics are:

- Maintainability (Analysability, Modifiability)
- Usability (Learnability, Operability)

5. ***Code Security*** evaluates security posture through analysis of various security and compliance related features such as sensitive data handling, security service configurations, and credential management. The related ISO characteristics are:

- Security (Confidentiality, Integrity, Non-repudiation, Accountability, Authenticity)

6. ***Functionality and Purpose*** assesses functional completeness and deployment readiness through verification of essential documentation, syntactic correctness, and execution behaviour. The related ISO characteristics are:

- Functional Suitability (Functional Correctness, Functional Completeness)
- Usability (Appropriateness Recognizability)

7. ***Error Handling*** measures resilience and fault tolerance through evaluation of error management strategies, recovery mechanisms, and debugging capabilities. The related ISO characteristics are:

- Reliability (Maturity, Fault Tolerance, Recoverability)
- Maintainability (Analysability)

8. *Code Automation* evaluates autonomous operation capabilities through variable management, execution mode support, and operational flexibility. The related ISO characteristics are:

- Performance Efficiency (Resource Utilization, Capacity)
- Usability (Operability)
- Reliability (Maturity)

9. *Code Integration* assesses interoperability and modularity capabilities within automation ecosystems, measuring integration with external systems and orchestration participation. The related ISO characteristics are:

- Compatibility (Interoperability, Co-existence)
- Portability (Adaptability)
- Maintainability (Modularity)

The nine categories presented are designed to accommodate quality attributes unique to IaC such as metadata indicators, automation capabilities, and integration requirements that traditional software quality models do not address.

Section 3.2.4 examines how the attributes from both catalogues map to the proposed categories and compares that mapping with their direct application to ISO/IEC 25010:2011. This comparison reports coverage of the published attributes rather than completeness across all possible IaC quality concerns.

3.2.4 Attribute Integration into the Framework

Demonstrating the proposed framework's capacity to incorporate existing research while addressing limitations identified in ISO/IEC 25010:2011 mapping requires comparative analysis. The comparative analysis examines how attributes from prior studies map to both the ISO standard and the proposed framework.

The comparison methodology involves mapping attributes identified by Rahman et al. [4] and Dalla Palma et al. [5] to the nine framework categories. This analysis categorises mappings as direct (attributes explicitly align with

category definitions), previously indirect (attributes influence category concerns without explicit alignment with ISO/IEC 25010:2011 sub-characteristics), or previously unmapped (attributes that found no representation in ISO/IEC 25010:2011). The comparison highlights how the framework accommodates attributes that the ISO standard cannot represent. Tables 3.7 and 3.8 present the complete mapping results, demonstrating the framework's coverage of IaC quality attributes.

Comparison of Rahman et al.'s Attribute Mappings

Table 3.7 presents the mapping of Rahman et al.'s 12 attributes to the proposed framework. The analysis reveals complete accommodation of all attributes, including those previously unmapped to ISO characteristics.

Table 3.7: Mapping of IaC Attributes from Rahman et al.'s [4] to Proposed Quality Framework

Note: "Indirect Mapping" and "No Mapping" columns show the previous mapping status from ISO/IEC 25010:201 (Table 3.3)

IaC Code Quality Category	Direct Mapping	Indirect Mapping (Previously)	No Mapping (Previously)
Metadata	×	×	×
Code Structure	File	Attribute, Lines of Code	×
Code Sophistication	Ensure	×	×
Code Maintainability	Comment	×	×
Code Security	SSH Key, File Mode	Hard-Coded String	×
Functionality & Purpose	×	×	×
Error Handling	×	×	×
Automation	×	×	URL, Commands
Code Integration	Include, Require	×	×

Legend:

× - No IaC code attributes maps to proposed IaC code quality category

The attributes "File," "Lines of Code," and "Attribute" map to Code Structure, capturing organisational aspects of infrastructure scripts. Security-related properties "SSH Key," "File Mode," and "Hard-Coded String" align with Code Security. The "Ensure" attribute maps to Code Sophistication, reflecting its role in state management. Previously unmapped attributes "URL" and "Commands" find representation in Code Automation, acknowledging their role in external resource management and system operations.

The framework's Metadata category remains unpopulated for Rahman et al.'s attributes, as their study focused exclusively on code-level properties. The absence reinforces the identified gap in existing research regarding metadata consideration. Similarly, categories such as Functionality and Purpose and Error Handling show no mappings, indicating areas not addressed in their analysis that focussed narrowly on defects.

Comparison of Dalla Palma et al.'s Attribute Mappings

Table 3.8 demonstrates the framework's capacity to accommodate all 46 attributes from Dalla Palma et al.'s comprehensive catalogue. The mapping reveals how previously unmapped infrastructure-specific constructs find appropriate categorisation within the framework.

Table 3.8: Mapping of Iac Attributes from Dalla Palma et al.'s [5] to Proposed Quality Framework

Note: "Indirect Mapping" and "No Mapping" columns show the previous mapping status from ISO/IEC 25010:201 (Table 3.4)

IaC Code Quality Category	Direct Mapping	Indirect Mapping (Previously)	No Mapping (Previously)
Metadata	×	×	×
Code Structure	File	Variables, Lines Blank, Lines Source Code, Blank Space Between Words, Avg Play Size, Avg Task Size, Length Tasks, Unique Names, Names with Variables	×
Code Sophistication	Loops	Entropy, Conditions, Decisions, Math Operations, Regex	Lookups, Keys, (External, Distinct) Modules
Code Maintainability	Comment	User Interaction	×
Code Security	SSH Key, File Mode	Suspicious Comments, Deprecated Keywords, Deprecated Modules	×
Functionality & Purpose	×	×	×
Error Handling	Ignore Errors	Blocks	×
Automation	Paths	×	Commands, URL, Filters, Play, Roles, Fact Module
Code Integration	Import Playbook, Import Role, Import Task, Include Role, Include Task, Include Vars, Include	×	×

Legend:

× - No IaC code attributes maps to proposed IaC code quality category

Code Structure incorporates the majority of organisational metrics including "Variables," "Lines Blank," "Avg Play Size," and "Unique Names." Code Sophistication captures advanced constructs such as "Loops," "Conditions," "Entropy," and previously unmapped elements including "Lookups," "Keys," and "Modules." The framework successfully categorises infrastructure-specific

attributes such as “Commands,” “URL,” “Filters,” “Play,” “Roles,” and “Fact Module” within Code Automation, addressing their operational nature.

Error Handling accommodates “Ignore Errors” and “Blocks,” while Code Integration encompasses all import and include mechanisms. The mapping demonstrates the framework’s ability to represent the full spectrum of Ansible quality attributes, including those that found no place within ISO/IEC 25010:2011.

Framework Coverage via Attribute Mapping

The comparative analysis demonstrates the framework’s design through several observations:

First, the mapping exercise demonstrates that all attributes from both studies find appropriate categorisation within the framework, eliminating the “no mapping” classification that occurred in ISO mapping. Previously unmapped attributes such as “URL,” “Commands,” “Plays,” “Roles,” and platform-specific modules align with framework categories designed to address infrastructure automation concerns.

Second, the framework’s categories group functionally related attributes coherently than ISO characteristics. Code Automation consolidates execution-related attributes (commands, URLs, filters, plays, roles, fact modules) that ISO scatters across Performance Efficiency, Usability, and Reliability. Similarly, Code Integration groups all import and include mechanisms that ISO distributes across Portability, Compatibility, and Maintainability. This consolidation provides clearer quality assessment dimensions aligned with how practitioners reason about IaC.

Third, the distribution of attributes across categories suggests quality assessment patterns. Code Structure and Code Sophistication receive numerous mappings, indicating their importance in existing research. The absence of mappings to Metadata confirms the identified research gap, while populated categories demonstrate alignment with established quality concerns.

The successful accommodation of all examined attributes demonstrates coverage of every attribute reported in existing research . Where ISO/IEC 25010:2011 does not represent infrastructure-specific properties, the proposed framework provides categories that maintain conceptual alignment with software quality principles while addressing IaC-specific concerns. This mapping capability suggests that the framework offers a more suitable foundation for IaC quality assessment than direct application of traditional software quality models.

Completeness of the Proposed Categories

The mapping establishes backward coverage. Every attribute reported by the two prior studies finds a category in the framework. Backward coverage does not by itself guarantee forward completeness, because attributes that no study has yet reported could in principle fall outside the nine categories. The distinction matters for a framework intended to remain useful as IaC research grows.

The expectation that the categories will absorb future attributes rests on their grounding. The framework extends the quality decomposition of ISO/IEC 25010:2011, which is intended to cover the space of software product quality, and introduces categories for the infrastructure-specific concerns that the standard does not represent. A future attribute must express some quality concern. Any concern within the scope of the standard is covered by construction, so the exposure narrows to concerns that lie outside both the standard and the current infrastructure-specific categories.

The thesis already exercises the scenario in question. The implementation extracts 96 attributes (Table 3.9), extending beyond the 46 attributes of Dalla Palma et al. [5]. The additions populate the Metadata and the Functionality and Purpose categories, in which neither mapped study reports any attribute. Every one of these previously unreported attributes found a category without any change to the category set. Chapter 7 later uses the same nine categories to

assess AI-generated IaC, a class of scripts that neither mapped study examined, and the categories accommodate this class without extension.

The framework accommodates growth in two ways. A new attribute that expresses an existing concern joins its category directly, because each category is defined by the concern it covers rather than by a fixed attribute list, and the scoring algorithm presented in Section 3.3 operates on whichever attributes an organisation selects. A concern that fits no existing category would instead require a new category. Categories are assessed independently and combined afterwards, so an added category leaves the assessment of existing categories unchanged. One limitation remains as a matter of principle. Completeness of a category set cannot be proven. It can only be falsified by an attribute that fits no category, and no such counter-example has emerged across the 96 extracted attributes or the applications in the chapters that follow.

3.3 Quality Score Computation Algorithm

This section presents an algorithm for computing quality scores from repository attributes. The algorithm consists of four parts that extract, normalise, weight, and aggregate quality metrics. Algorithm 3.3 presents the complete computational procedure.

The algorithm adopts a relative assessment approach rather than absolute measurement. This design decision reflects a practical constraint in IaC quality evaluation. No universally accepted reference repository exists that could serve as a theoretical benchmark for IaC software quality. Consequently, the algorithm compares repositories within an organisation's collection against each other.

Algorithm 3.3: Quality Score Computation

Set of repositories $R = \{r_1, ..., r_n\}$,
Organisational policy defining the following:
  $A_c = \{a_0, a_1, a_2, \ldots, a_{k_c}\}$ ▷ Selected attributes for category c
  $\tau_j \in \{+1, -1\}$ for each $j \in A_c$ ▷ Policy-assigned type indicator
  $P_c = \{j \in A_c : \tau_j = +1\}$, $N_c = \{j \in A_c : \tau_j = -1\}$
  $C_{active} = \{c : |A_c| > 0\}$ ▷ Active categories
Part 1: Attribute Extraction
for each repository $r_i \in R$ **do**
  for each category $c \in C_{active}$ **do** ▷ Only process active categories
    Extract raw values for attributes in A_c:
      $A_{i,c} = [a_{i,c,1}, ..., a_{i,c,k_c}]$ where $k_c = |A_c|$
Part 2: Attribute Normalisation
for each category $c \in C_{active}$ **do**
  for each attribute $j \in \{1, ..., k_c\}$ **do**
    $V_{c,j} = [a_{1,c,j}, ..., a_{n,c,j}]$
    $m \leftarrow \text{median}(V_{c,j})$; $v_{min} \leftarrow \min(V_{c,j})$; $v_{max} \leftarrow \max(V_{c,j})$
    for each repository $i \in \{1, ..., n\}$ **do**
      $d_i \leftarrow a_{i,c,j} - m$
      if $a_{i,c,j} < m$ and $m > v_{min}$ **then** ▷ Condition guard
        $n_{i,c,j} \leftarrow \frac{d_i}{|v_{min} - m|}$ ▷ Normalise to [-1,0]
      else if $a_{i,c,j} > m$ and $v_{max} > m$ **then** ▷ Condition guard
        $n_{i,c,j} \leftarrow \frac{d_i}{v_{max} - m}$ ▷ Normalise to [0,1]
      else
        $n_{i,c,j} \leftarrow 0$ ▷ At median → 0
      $n_{i,c,j} \leftarrow \frac{n_{i,c,j}+1}{2}$ ▷ Final mapping to [0,1]
      if $j \in N_c$ **then** ▷ Check if attribute j is in negative set
        $n_{i,c,j} \leftarrow 1 - n_{i,c,j}$
        if $a_{i,c,j} = v_{min}$ **then** ▷ Cleanest value → full score
          $n_{i,c,j} \leftarrow 1.0$
Part 3: Policy-Based Weighting
for each repository r_i **do**
  for each category $c \in C_{active}$ **do** ▷ Only score active categories
    $W_c = [w_{c,1}, ..., w_{c,k_c}]$ ▷ Defined by organisational policy
    $W_{total} = \sum_{j \in P_c} w_{c,j} + |N_c|$ ▷ Sum of P_c weights + count of N_c
    $S_{i,c} = \frac{1}{W_{total}} \left(\sum_{j \in P_c} w_{c,j} \cdot n_{i,c,j} + \sum_{j \in N_c} n_{i,c,j} \right)$ ▷ $S_{i,c} \in [0, 1]$,
negative attrs have implicit weight 1.0
Part 4: Score Aggregation
Category weights $W^{cat} = \{w_c^{cat} : c \in C_{active}\}$ ▷ Defined by org's policy
$S_{max} = \sum_{c \in C_{active}} w_c^{cat}$ ▷ Maximum score
for each repository r_i **do**
  $S_{total}(i) = \sum_{c \in C_{active}} w_c^{cat} \cdot S_{i,c}$ ▷ $S_{total}(i) \in [0, S_{max}]$
  $S_{alignment}(i) = \frac{S_{total}(i)}{S_{max}} \times 100$ ▷ $S_{alignment}(i) \in [0, 100]\%$

This relative approach suits deployment scenarios where organisations manage multiple IaC repositories and require quality assessment across their portfolio. To accommodate varying organisational priorities, the algorithm accepts policy configurations that encode high-level strategies and development methodologies. These policies determine which attributes the algorithm evaluates and how it weights them. For example, an organisation focused on compliance-driven infrastructure would configure different weights than one prioritising security smell detection. The following subsections describe each component of the algorithm in detail.

3.3.1 Attribute Extraction

The algorithm begins by defining the organisational policy that determines attribute selection and categorisation. For each category $c \in \{1, \ldots, 9\}$ corresponding to the nine quality categories (Metadata, Code Structure, Code Sophistication, Code Maintainability, Code Security, Functionality and Purpose, Error Handling, Code Automation, and Code Integration), the organisation specifies an set of selected attributes $A_c = \{a_0, a_1, a_2, \ldots, a_{k_c}\}$, where k_c denotes the number of attributes selected for category c and each element $j \in A_c$ identifies a selected attribute.

To ground these categories in concrete measurement, Table 3.9 presents the 96 attributes mapped across the nine categories. These attributes were derived through static analysis of Ansible playbooks, tasks, handlers, and variable files, expanding upon the 46 attributes identified by Dalla Palma et al. [5] to include both code-level metrics and metadata indicators. The extraction employs static analysis of Ansible roles, tasks, handlers, and variable files to identify keywords and structural patterns, which are categorised according to their operational roles within the official Ansible documentation. Symbols ($\star$, $*$, $\curlywedge$, $\uplus$, $\bowtie$) adjacent to some attributes represent aggregated calculations, as the underlying attributes are defined operationally in a similar manner. The aggregation avoids double-counting and ensures related attributes contribute

appropriately to category scores without over-inflating quality measures with redundant metrics. For the detection of deprecated keywords and modules, a manual review of Ansible compiler changelogs across versions was performed to compile a comprehensive list of 837 unique deprecated keywords and modules, enabling accurate detection of obsolete components during repository analysis.[11]

Table 3.9: Mapping of Code Attributes to Proposed Code Quality Framework Categories

IaC Code Quality Category	Code Attributes
Metadata	Download count, Tag, Total versions, Dependencies, Supported platform count, Stars, Forks, Version release time, Github status, Minimum Ansible versions, Average update time, Open issues
Code Structure	Count of lines in YAML/YML files, Lines of code in YAML/YML files, Task count, Count of directories, Variables, Files count, Lines in source code, Blank spaces between words, Average play size, Average task size, Unique names, Names with variables, Length of task, Template files, Line blanks
Code Sophistication	Entropy, Loops, Conditions, Decisions, Include, Ensure, Keys, All modules, Mathematical operations, Regex, Lookups
Code Maintainability	README word count, Comments, User interactions
Code Security	SELinux, Firewalld, Vault, stat, File modes, No Log, apt_key, SSH keys, Password, Suspicious comments, Deprecated keywords, Deprecated modules
Functionality & Purpose	README count, Licence file presence, Syntax Check
Error Handling	failed_when, changed_when, $Rescue^{\star}$, $Always^{\star}$, $Block^{\star}$, Debug, any_errors_fatal settings, max_fail_percentage, $Retry^{\star}$, $Until^{\star}$, $Delay^{\star}$, ignore_errors
Code Automation	Become, vars, check_mode, Paths, Plays, Roles, $URI^{\curlywedge}$, $URL^{\curlywedge}$, Fact modules, Filters, environment, local_action, Fetch modules, Handlers, Tags, Commands
Code Integration	wait_for, Git, Import playbooks, Import $roles^{\uplus}$, Include $roles^{\uplus}$, Import $tasks^{\bowtie}$, Include $tasks^{\bowtie}$, Include vars, Rsync, win_service, add_host

Legend: Symbols ($\star$, $*$, $\curlywedge$, $\uplus$, $\bowtie$) indicate grouped attributes to prevent double counting

This attribute set enables evaluation of quality while maintaining computational tractability. The metadata attributes provide context for interpreting the quality of code-level attributes, addressing the gap found in previous research that focused solely on the properties of code. The code attributes measure both the structural and behavioural aspects of code, enabling assessment

[11] Ansible Community, "Ansible Release and Maintenance," 2025, https://docs.ansible.com/ansible/latest/reference_appendices/release_and_maintenance.html.

of static properties. The classification of these attributes into mandatory practices, recommended patterns, and discouraged anti-patterns for policy-based weighting is discussed in Chapter 4.

The policy assigns each attribute $j \in A_c$ a type indicator $\tau_j \in \{+1, -1\}$ that encodes how the attribute relates to quality under that policy. When higher values of attribute j indicate better quality, $\tau_j = +1$ (positive attribute). When higher values indicate worse quality, $\tau_j = -1$ (negative attribute). This classification is policy-dependent rather than intrinsic to the attribute. For instance, the dependency count attribute receives $\tau_j = +1$ under the Ansible best-practice guidelines, where dependencies indicate integration maturity. The same attribute receives $\tau_j = -1$ under a supply chain detection context, where dependencies introduce external trust relationships that propagate security risks. This reclassification demonstrates that τ_j is determined by the policy's assessment objective rather than by any fixed property of the attribute itself. The resulting sets P_c and N_c (Algorithm 3.3, line 5) partition the selected attributes such that $P_c \cup N_c = A_c$ and $P_c \cap N_c = \emptyset$. The algorithm identifies active categories as $C_{active} = \{c : |A_c| > 0\}$, enabling organisations to focus assessment on relevant quality dimensions.

For each repository r_i in the repository set $R = \{r_1, ..., r_n\}$, the extraction process operates only on active categories $c \in C_{active}$, generating attribute array $A_{i,c} = [a_{i,c,1}, ..., a_{i,c,k_c}]$ where $k_c = |A_c|$ represents the number of selected attributes in category c. Each element $a_{i,c,j}$ represents the raw value of attribute j in category c for repository i.

This policy-driven extraction provides organisations with flexibility to tailor quality assessment to their specific goals and practices. Organisations select which attributes to include in each category based on their quality priorities. A security-focused organisation might emphasise attributes in the Code Security category while a team prioritising long-term maintenance might emphasise Code Maintainability attributes. The framework accommodates these varying priorities without requiring modification to the underlying algorithm. Further-

more, organisations can adjust their policies over time as their practices evolve or as new concerns emerge. This configurability enables the same algorithmic framework to serve diverse organisational contexts ranging from compliance-driven enterprises to agile development teams.

3.3.2 Attribute Normalisation

The normalisation phase uses medians to scale attribute values, handling outliers and ensuring comparability across repositories. Section 3.3.2.2 justifies this choice against the conventional min–max alternative. For each attribute j (where $j \in \{1, ..., k_c\}$ represents the attribute index within a category) in category c, the algorithm constructs a vector $V_{c,j} = [a_{1,c,j}, ..., a_{n,c,j}]$, which contains all repository values for that specific attribute across the entire dataset. Here, $V_{c,j}$ represents the complete vector of values for attribute j in category c, aggregating data from all n repositories.

The algorithm then computes three statistical measures: m representing the median of the attribute values in $V_{c,j}$, v_{min} denoting the minimum value in the attribute vector, and v_{max} indicating the maximum value. These measures establish the distribution characteristics necessary for normalisation.

For each repository value $a_{i,c,j}$, the distance from median is calculated as $d_i = a_{i,c,j} - m$, where d_i represents the signed distance of repository i's value from the median for that attribute. The normalisation then proceeds conditionally to produce $n_{i,c,j}$, the normalised attribute value that will ultimately fall within the range $[0, 1]$:

- If $a_{i,c,j} < m$ and $m > v_{min}$: The value normalises to $n_{i,c,j} = \frac{d_i}{|v_{min} - m|}$, mapping to the interval $[-1, 0]$
- If $a_{i,c,j} > m$ and $v_{max} > m$: The value normalises to $n_{i,c,j} = \frac{d_i}{v_{max} - m}$, mapping to the interval $[0, 1]$
- If $a_{i,c,j} = m$: The normalised value equals 0

Following initial normalisation, values are remapped to $[0, 1]$ using the transformation $n_{i,c,j} = \frac{n_{i,c,j}+1}{2}$. For negative attributes (N_c) (an attribute type indicator signifying that lower raw values represent better quality, such as security vulnerabilities or deprecated modules), the algorithm applies the inversion $n_{i,c,j} = 1 - n_{i,c,j}$. Repositories with minimum negative attribute values receive a perfect score of 1.0 for those attributes.

The use of medians for normalisation provides robustness against outliers common in repository metrics. Consider the attribute "number of tasks" measured across five repositories with values 10, 12, 15, 18, and 200. The fifth repository represents an enterprise-scale deployment that functions as an outlier. The mean of these values is 51, while the median is 15. If the algorithm used mean-based normalisation, the four repositories with values between 10 and 18 would all appear below average despite representing typical IaC implementations. The outlier would dominate the distribution and distort the assessment of all other repositories. By using the median as the central reference point, the algorithm evaluates repositories against a value of 15, which represents the typical case. The outlier repository is still recognised as above the median, but it does not distort the relative positioning of other repositories. This approach ensures that quality assessment reflects typical repository characteristics rather than being skewed by extreme cases. The separation of positive and negative attributes ensures correct quality interpretation across both attribute types.

Table 3.10 traces the same five repositories through Part 2 of Algorithm 3.3. The statistics for this attribute vector are $m = 15$, $v_{min} = 10$, and $v_{max} = 200$. Values below the median divide by $|v_{min} - m| = 5$ and values above the median divide by $v_{max} - m = 185$. After the remapping step, the minimum receives 0, the median receives 0.5, and the maximum receives 1. The value 18 lies close to the median and receives 0.508, because the outlier at 200 stretches the upper interval, so the upper half of the scale is reserved for repositories that are well above the typical case. The fifth column is the final score when the attribute

is positive, since no further transformation applies after the remap. The sixth column is the final score when the same attribute is negative, obtained by inverting the remapped value. The inversion reverses the ordering, and the repository at v_{min} receives 1.0 through the final override in the algorithm.

Table 3.10: Worked Example of Attribute Normalisation for the Values 10, 12, 15, 18, and 200

Raw value $a_{i,c,j}$	$d_i = a_{i,c,j} - m$	Branch	Initial $n_{i,c,j}$	Final score, positive attribute	Final score, negative attribute
10	-5	below median	$-5/5 = -1.000$	0.000	1.000
12	-3	below median	$-3/5 = -0.600$	0.200	0.800
15	0	at median	0	0.500	0.500
18	$+3$	above median	$3/185 = 0.016$	0.508	0.492
200	$+185$	above median	$185/185 = 1.000$	1.000	0.000

3.3.2.1 Special Cases in the Normalisation

The median based normalisation handles three degenerate distributions that arise when the same value appears many times. The first is $m = v_{min}$, which occurs when at least half of the repositories share the minimum value. Count-based attributes commonly produce this case, because most repositories record zero for attributes such as deprecated modules or template files. No value can then lie below the median, so the first branch never executes, and the guard $m > v_{min}$ ensures that the divisor $|v_{min} - m|$ can never be zero. All repositories at the shared minimum equal the median and receive 0.5 after remapping. The second case is $v_{max} = m$, the mirror situation in which at least half of the repositories share the maximum. The guard $v_{max} > m$ protects the upper divisor in the same way, and values at the shared maximum receive 0.5.

The third case is a constant vector, where $v_{min} = m = v_{max}$ and every repository holds the same value. Every value then routes to the final branch and receives 0.5 after remapping. For a positive attribute this is the intended outcome, because an attribute that does not vary carries no information for relative assessment, and 0.5 is the neutral score. For a negative attribute the override in Algorithm 3.3 then raises every repository to 1.0, because in a con-

stant vector every value equals v_{min}. This asymmetry is deliberate. A constant negative attribute means the anti-pattern is absent from every repository in the collection, and the absence of an anti-pattern merits full credit rather than a neutral score.

The override serves a broader purpose than the constant case. For negative attributes where more than half of the repositories are clean, such as security smells, the median equals the minimum, and a repository with no occurrences would otherwise receive only the neutral 0.5. The override guarantees that the cleanest observed repositories receive the full score of 1.0. The cost is a discontinuity: a repository with a single occurrence scores below 0.5 while a repository with none scores 1.0. This reflects the same zero-tolerance position applied to security assessment in Chapter 6, where the presence of any smell is treated as qualitatively different from its absence.

3.3.2.2 Comparison with Min–Max Normalisation

A simpler alternative would normalise each value directly with $n_{i,c,j} = \frac{a_{i,c,j} - v_{min}}{v_{max} - v_{min}}$, which also maps every value to $[0, 1]$. Reaching the unit interval is therefore not the reason for the median-based design. The reason is discrimination. Repository metrics such as download counts, line counts, and task counts are heavily right-skewed, so almost every attribute vector contains a small number of extreme repositories far above the bulk. Min–max normalisation anchors the scale to those two extremes and therefore spends most of the unit interval on the gap between the bulk and the outliers, compressing the typical repositories, which are the ones an assessment must tell apart, into a narrow band at the lower end. The median-based scheme anchors the scale to the typical repository instead, so the typical range occupies half the interval and differences among ordinary repositories remain visible. Two further properties follow from the same anchoring: interpretability, because the score 0.5 always denotes the typical repository rather than an arbitrary point, and stability, because the anchor is insensitive to extreme values joining the collection.

Table 3.11 applies both schemes to the worked example of Table 3.10. Under min–max normalisation the four typical repositories are compressed into four per cent of the scale by the single outlier, and at two decimal places they are close to indistinguishable; the median repository itself receives 0.026, a score with no fixed meaning. Under the median-based scheme the same four repositories spread from 0 to 0.508 and the median repository receives exactly 0.5. The outlier in this example represents the normal operating condition for repository metrics rather than a corner case.

Table 3.11: Comparison of Min–Max and Median-Based Normalisation for the Values 10, 12, 15, 18, and 200

Raw value $a_{i,c,j}$	Min–max score $\frac{a_{i,c,j}-v_{min}}{v_{max}-v_{min}}$	Median-based score (Algorithm 3.3)
10	$0/190 = 0.000$	0.000
12	$2/190 = 0.011$	0.200
15	$5/190 = 0.026$	0.500
18	$8/190 = 0.042$	0.508
200	$190/190 = 1.000$	1.000

The stability difference appears as the collection grows. A min–max scale is anchored to the extremes, so a single new extreme repository changes the divisor $v_{max} - v_{min}$ and with it every existing score. The median is determined by the middle of the ordered values, so a single new repository shifts it by at most one position regardless of how extreme the new value is, and existing scores remain close to their previous values.

3.3.3 Policy-Based Weighting

The category scoring phase incorporates organisational policies through attribute weights applied only to active categories identified in Part 1. For each active category $c \in C_{active}$, a weight vector $W_c = [w_{c,1}, ..., w_{c,k_c}]$ defines the relative importance of positive attributes, where $w_{c,j}$ represents the weight assigned to attribute j within category c as defined by organisational policy. The total weight calculation accounts for both explicitly weighted positive attributes and normalised negative attributes:

$$W_{total} = \sum_{j \in P_c} w_{c,j} + |N_c|$$

where $|N_c|$ represents the count of negative attributes in category c. This design ensures that negative attributes contribute with an implicit weight of 1.0 after their normalisation inversion in Section 3.3.2. The separation prevents double-counting their negative impact while maintaining their influence on quality assessment.

The category score $S_{i,c}$ for repository r_i in category c is computed as:

$$S_{i,c} = \frac{1}{W_{total}} \left(\sum_{j \in P_c} w_{c,j} \cdot n_{i,c,j} + \sum_{j \in N_c} n_{i,c,j} \right)$$

where $S_{i,c}$ represents the category quality score constrained to the range $[0, 1]$, indicating the normalised quality performance of repository i in category c. The formula explicitly separates the contribution of positive attributes (weighted by organisational policy) from negative attributes (contributing with unit weight after normalisation). Because the numerator and W_{total} scale together, only the ratios between weights affect $S_{i,c}$; the choice of specific values is discussed in Section 4.3.2.

For inactive categories where $c \notin C_{active}$, the algorithm assigns $S_{i,c} = 0$, ensuring that unselected quality dimensions do not influence the assessment. This approach enables organisations to encode development priorities and compliance requirements through explicit attribute selection and weighting while focusing assessment on relevant quality dimensions. Detailed application of policies and their weights are discussed in the following Chapter 4.

3.3.4 Score Aggregation

The final phase aggregates category scores into a quality metric using policy-defined category weights. Organisations define category weights $W^{cat} = \{w_c^{cat} : c \in C_{active}\}$ as part of their policy configuration, where each w_c^{cat} represents the importance of category c to the organisation. The algorithm first calculates

the maximum achievable score for the given policy:

$$S_{max} = \sum_{c \in C_{active}} w_c^{cat}$$

For each repository r_i, two complementary scores are computed. The total quality score:

$$S_{total}(i) = \sum_{c \in C_{active}} w_c^{cat} \cdot S_{i,c}$$

Which represents the weighted sum of category scores, yielding values in $[0, S_{max}]$. To enable meaningful comparison of repositories across different policies, the algorithm computes an alignment percentage:

$$S_{alignment}(i) = \frac{S_{total}(i)}{S_{max}} \times 100$$

where $S_{alignment}(i) \in [0, 100]\%$ indicates the percentage of maximum quality achieved within the organisation's chosen policy. These two scores serve different purposes. The total score (S_{total}) provides an absolute performance metric for within-policy comparison, while the alignment percentage ($S_{alignment}$) enables cross-policy evaluation.

The computational framework presented through Algorithm 3.3 establishes a foundation for policy-driven IaC quality assessment, transforming raw repository attributes into meaningful quality indicators that reflect organisational priorities while maintaining comparability across different development practices .

3.3.5 Practical Deployment Considerations

The relative design of the algorithm raises a practical question. When a new repository arrives with an attribute value above the existing v_{max} or below the existing v_{min}, how is it scored, and must the existing scores be recomputed?

The implementation recomputes the statistics vector (m, v_{min}, and v_{max}

for every attribute) on each scoring run over the current repository collection. A new repository is scored by adding it to the collection and rerunning the normalisation, so an out-of-range value cannot arise. A repository that sets a new extreme for an attribute defines the new v_{max} or v_{min} and receives the boundary score for that attribute. Recomputation is not expensive because the dominant cost is attribute extraction, which runs once per repository and can be cached. Only the normalisation, weighting, and aggregation stages rerun, and these are linear in the number of repositories for each attribute and linear in the number of attributes for each repository. Scores of existing repositories can shift when the statistics change, which is a property of the relative assessment design rather than a defect, because each score expresses a position within the current population.

Part of the refresh can be performed incrementally. When a repository is added, the minimum (v_{min}) and maximum (v_{max}) of an attribute can be updated with a single comparison. The median cannot, because computing a new median requires the full set of stored values. The statistics are therefore refreshed by recomputing them over the whole collection, and this remains a low-cost operation because the computation is linear. In practice much of the range is stable across refreshes. Of the attribute metrics extracted by the implementation, 80 are counts with a natural minimum of zero, so v_{min} cannot decrease for them. Between refreshes, scores change mainly through gradual movement of the median rather than through changes to the range. The evaluation in Chapter 4 applied this procedure to 26,986 repositories, demonstrating that the approach operates at ecosystem scale .

3.4 Chapter Summary

This chapter presented a framework for measuring IaC script quality, addressing limitations in current assessment practices and software quality standards. Investigation of quality evaluation mechanisms across IaC platforms revealed

heterogeneity, with configuration management tools implementing scoring systems while infrastructure provisioning and image building tools lacking assessment methods. Mapping IaC code attributes from existing research to ISO/IEC 25010:2011 demonstrated gaps, with infrastructure-specific properties identified by Rahman et al. [4] and Dalla Palma et al. [5] having no correspondence in the ISO standard. The proposed framework addresses these gaps through nine categories. These categories are Metadata, Code Structure, Code Sophistication, Code Maintainability, Code Security, Functionality and Purpose, Error Handling, Code Automation, and Code Integration. Each category extends ISO/IEC 25010:2011 characteristics to accommodate infrastructure-specific concerns. All attributes from previous studies found categorisation within the framework, supporting its design. The framework is supported by Algorithm 3.3, which computes quality scores through a four-part process encompassing extraction, normalisation, policy-based weighting, and score aggregation. The algorithm uses medians to normalise attribute values, reducing the influence of outliers in quality assessment. Organisations can configure which attributes to assess and how to weight them according to their priorities. The framework compares repositories against each other rather than against theoretical benchmarks, enabling quantitative comparison across repositories within an organisation.

As noted in Section 1.3, we use ISO/IEC 25010:2011 for the mapping and analysis, which was superseded by ISO/IEC 25010:2023 during the course of this research. The 2023 revision introduces three changes relevant to this work. First, the addition of a safety as a characteristic. However, safety assessment lacks implementation support in current IaC tooling, and the framework's Error Handling and Code Security categories already capture the fault tolerance and risk mitigation concerns that the safety characteristic addresses in traditional software contexts. Second, the renaming of "usability" to "interaction capability" and "portability" to "flexibility" constitutes terminology changes that do not alter the underlying sub-characteristic definitions to which

this chapter's mapping refers. Third, no new sub-characteristics pertaining to state management, idempotency, configuration drift, or repository metadata were introduced. The gaps identified between the 2011 edition and IaC requirements therefore persist in the 2023 edition, and the nine proposed categories remain necessary to address these gaps.

Chapter 4

Policy-Driven Quality Assessment and Empirical Analysis

Assessing IaC quality requires systems that allow organisations to prioritise their needs and meet their development standards. As described in Chapter 3, our proposed methodology for assessing IaC quality through nine categories with associated attributes can be made organisation-specific using policies to weigh the relative importance of each category and attribute.

This chapter identifies how policy configurations, expressed as weights, transform the framework into an organisation-specific evaluation tool. The chapter first establishes the theoretical basis of policy-based assessments by showing how organisational priorities are translated into quantifiable evaluation criteria. It then explores how the framework implements policy through weight vectors at two hierarchical levels: attribute-level weights, which represent best practice for technical considerations, and category-level weights, which represent broader development philosophies. Following this, the chapter presents examples of specific policy configurations, ranging from baseline equal-weight methods to best-practice configurations derived from community standards, and finally use-case configurations that reflect the development ap-

proaches of various organisational environments. These use-case configurations are referred to as development philosophies throughout this chapter, as each represents a distinct set of principles that guide how an organisation prioritises quality in its infrastructure code. The chapter concludes by demonstrating the policy mechanism through the application of both attribute-level and category-level configurations to a large dataset of real-world IaC repositories, identifying how differing policy configurations affect the outcome of quality assessments and what patterns emerge in the quality of IaC repositories over time.

4.1 The Role of Policy in Software Quality Assessment

In software engineering, the term policy refers to formalised sets of rules or guidelines that define what is considered acceptable or unacceptable in terms of development practices and quality standards [61]. These policies define the parameters for quality assessment by establishing common standards for all development work. In an assessment context, policy manifests as evaluation criteria, metric thresholds, and weights assigned to different metrics, representing the organisational values and operational requirements encoded into each metric [62].

Traditional software quality frameworks, such as ISO/IEC 25010:2011 [12] and CMMI [63], are generally not designed to facilitate the incorporation of organisational policies into quality assessment practices. Although ISO/IEC 25010:2011 recognises variability in quality importance across organisations and permits organisations to select those quality characteristics and sub-characteristics that are most relevant to them, the standard does not provide substantial guidance on how to implement this customisation. Organisations are permitted to develop their own measurement methods and to assign priority to quality characteristics based on their requirements, but there are no explicit mechanisms for converting these priorities into numerical quality assessments. This gap

between the conceptual freedom to customise and the practical difficulty in implementing such customisations limits organisations' ability to link their quality assessments to their policies. Similarly, CMMI prescribes maturity levels and process areas with limited mechanisms for adjusting relative importance based on organisational context.

The need for policy-based quality assessments stems from the reality that differences exist across organisational contexts. For example, organisations in regulated industries are concerned primarily with security and compliance, while research institutions prioritise flexibility to experiment. Organisations involved in financial transactions have strict requirements for reliability and error tolerance that do not apply to those that rely on rapid deployment of new products. Policy-based quality assessments accommodate this diversity through customisable quality assessment tools that adapt generic quality models to fit a particular organisational context.

To address this need, our approach aims to enable organisations to express their priorities quantitatively through weighting assignment, threshold specification, and metric selection, facilitating the conversion of general quality standards into practical assessment standards. The goal of this approach is to provide a mechanism through which organisations can configure the framework to reflect their specific requirements without modifying the underlying assessment algorithm. Sections 4.2 and 4.3 detail how this approach is implemented within the IaC quality framework, first describing the policy encoding mechanism through weight vectors and then presenting specific policy configurations at both the attribute and category levels.

4.2 Policy Implementation in the IaC Quality Framework

The translation of organisational policies into quantitative assessment requires mechanisms within the IaC quality framework. While Section 3.3 established

the computational foundation through weighted attribute aggregation, this section examines the conceptual underpinnings and practical considerations of policy implementation within the framework context.

4.2.1 Policy Encoding Through Weight Vectors

The framework implements policy through weight vectors $W_c = [w_{c,1}, ..., w_{c,k}]$ defined for each quality category c, as introduced in Part 3 of Algorithm 3.3 and Section 3.3.3. These vectors encode organisational priorities by assigning relative importance values $w_{c,j}$ to each attribute j within a category. The mechanism transforms abstract policy statements into quantitative parameters that influence quality score computation. For example, an organisation that mandates secure credential management would assign a weight of $w_{c,j} = 1.0$ to the vault usage attribute within the Code Security category, while assigning a weight of $w_{c,j} = 0.5$ to recommended but non-mandatory attributes such as SSH key management. Additionally, Part 4 of the algorithm extends this policy encoding to the category level through category-level weights $W^{cat} = \{w_c^{cat} : c \in C_{active}\}$, enabling organisations to emphasise entire quality dimensions according to their development practices.

The weight-based encoding serves two functions within organisational contexts. First, it translates qualitative policy directives into numerical representations. When an organisation mandates "security-first development", this directive manifests through elevated weights for security-related attributes across relevant categories. Second, it enables policy evolution without architectural modifications. When organisational priorities change, the weight values can be adjusted without altering the underlying assessment algorithm. The following subsection describes how the framework structures these weight-based policies across two hierarchical levels.

4.2.2 Multi-Level Policy Configuration

The framework's policy implementation operates on a two level hierarchy that reflects the nature of managing IT infrastructure. The levels of this hierarchy determines both where and how weight-based quality policies are applied, establishing the bounds and degree of granularity for expressing quality policy, from organisation-wide standards to application or context-specific policy adjustments.

Attribute-Level Policy: At the most granular level of policy specification, organisations may apply weights to individual code quality attributes within categories. This level allows organisations to exercise fine-grained control over which quality factors they consider important. For example, within the Code Security category, an organisation operating in a cost-constrained environment with no sensitive data may assign a reduced weight to vault usage, as the overhead of implementing encrypted credential storage is not justified when all variables are non-sensitive. In contrast, an organisation handling financial data would assign a mandatory weight to the same attribute, reflecting the regulatory requirement to protect credentials.

Category-Level Policy: Organisations define category weights (W^{cat}) that reflect their philosophical orientation towards developing high-quality applications, as well as the intended use of those applications, in order to emphasise entire quality dimensions. A financial services organisation may assign higher weights to the Security and Error Handling categories, whereas a provider of software automation tools may assign higher weights to the Code Sophistication and Automation categories. This approach to defining quality policies enables organisations to strategically emphasise quality in a manner that is easily computable. This approach also allows the computation of an alignment score ($S_{alignment}(i)$), which measures the degree to which a repository i conforms to the organisation's stated quality priorities as encoded through its category-level weight configuration. The alignment score is computed as the weighted average of category scores, where the weights reflect the organi-

sation's category-level policy. This enables organisations to make meaningful comparisons across repositories, as each repository is evaluated against the same organisational weighting scheme.

4.3 Attribute-Level Policy Configurations

Policy configuration is an important component of the proposed framework, as it provides the means for empirically demonstrating how effectively the framework can encode an organisation's priorities. This section describes two attribute-level policy configurations, each serving a different purpose in the usage of our framework: a baseline (equal weights) policy provides a neutral and unbiased reference point against which to compare other policies, while a best-practice (domain-specific weight configuration) policy demonstrates how to configure the weights for a particular domain, and serves as an example of how organisations can apply the same process to derive their own policy configurations.

4.3.1 Equal-Weight Baseline Policy Configuration

A baseline policy using equal weights across all categories and attributes provides a neutral comparison of quality scores across repositories. By assigning equal weights, no single attribute is prioritised over another within the assessment process.

When a baseline policy is configured with equal weights, $w_{c,j} = 1.0$ is assigned for all positive attributes $j \in P_c$ in each category c and negative attributes $j \in N_c$ receive a weight of 1.0 after normalising inversion. Whether each attribute is classified as positive or negative is determined by the framework based on established community standards and platform documentation, as described in Section 4.5.3. Organisations can control the weights assigned to each attribute. When this configuration is used for calculating category scores the computation simplifies to:

$$S_{i,c}^{equal} = \frac{1}{k} \sum_{j=1}^{k} n_{i,c,j}$$

where $k = |P_c| + |N_c|$.This expression calculates the arithmetic mean of the normalised values of each attribute in the category, providing a means to compare scores across categories and assess the contributions of individual attributes to category assessment.

Regardless of the type of attribute being assessed (e.g., quality enhancers or quality detractors), the same equal-weight configuration is applied. Quality detractors are classified as negative attributes in N_c, and the framework handles their inverse relationship to quality through the normalisation process described in Algorithm 3.3, Part 2, in which they are inverted after normalisation ($(n_{i,c,j} = 1 - n_{i,c,j})$). In this way, the policy mechanism focuses solely on the relative importance of each attribute, while the normalisation mechanism handles the directionality of quality.

For category-level aggregation, the baseline policy uses $w_c^{cat} = 1.0$ for each of the nine categories, so no single quality dimension is given preference. The use of equal weights at both the attribute and category levels therefore provides a completely neutral baseline for assessing quality, against which other policy configurations can be compared.

4.3.2 Best-Practice Policy Configuration

While equal weighting provides an unbiased baseline, incorporating domain expertise through best-practice-based weights demonstrates the framework's capacity to encode community knowledge. To derive best-practice weights for Ansible, we analysed official documentation, established guidelines and community standards specific to the platform [1] [2] [3]. We drew from three

[1] Ansible Project, "Ansible Best Practices – Ansible Documentation," 2019, https://docs.ansible.com/ansible/2.8/user_guide/playbooks_best_practices.html.

[2] Red Hat Community of Practice, "Automation Good Practices," 2024, https://redhat-cop.github.io/automation-good-practices/.

[3] Ansible Project, "Ansible Tips and Tricks – Ansible Documentation," 2024, https://docs.ansible.com/ansible/latest/tips_tricks/ansible_tips_tricks.html.

categories of:

- Official platform documentation on recommended implementation practices
- Collective expertise reflected in community guides
- Industry standards for managing infrastructure requirements

Converting this qualitative guidance into quantifiable weights is achieved by assigning categorical values based on the nature of the guideline:

1. ***Mandatory practices*** *($w_{c,j} = 1.0$)*: Practices described as non-negotiable in official documentation are assigned the highest positive weight, including mandatory security controls, structural requirements, and necessary functional elements.
2. ***Recommended practices*** *($w_{c,j} = 0.5$)*: Practices that provide benefit to the IaC but are subject to contextual interpretation are assigned a reduced positive weight. In contrast to mandatory practices, these are described in official documentation as should "rather than must", implying there may be legitimate reasons to exclude them from the final solution.
3. ***Discouraged practices***: Practices that represent anti-patterns or security smells are assigned a negative flag value which, after normalisation, causes them to inversely affect the overall quality score. Official documentation typically indicates discouraged practices as sources of increased maintenance costs, security risks, or operational failures.

The assignment of each attribute to one of these three classes followed a documentation-driven decision rule. For every attribute, we located the statements in the sources listed above that describe the practice the attribute measures, and we classified the attribute according to the strength of the language used in those statements. Guidance stated as an obligation, using terms

such as *must* or *never* with no stated exceptions, produced a mandatory classification. Guidance stated as advice, using terms such as *should* or *prefer*, or describing the conditions under which the practice is beneficial, produced a recommended classification. Guidance that warns against a practice, lists it as an anti-pattern, or marks it as deprecated produced a discouraged classification. Where the sources differed in strength for the same practice, the strongest statement determined the class.

The following examples illustrate how the rule assigned one attribute from each class:

- ***Vault (mandatory,*** $w_{c,j} = 1.0$***)***: The Vault attribute captures the use of Ansible Vault for encrypted storage of sensitive values. Official documentation advises encrypting sensitive or secret variables with Ansible Vault [4], and the community guidance instructs users never to store passwords or other sensitive values in plain text in version-controlled files [5]. Taking the strongest statement, the requirement to protect credentials admits no exception, and Ansible Vault is the platform mechanism for meeting it within role content, so the attribute is classified as mandatory.

- ***Handlers (recommended,*** $w_{c,j} = 0.5$***)***: The Handlers attribute captures the use of handler tasks that run only when notified of a change. Official documentation introduces handlers with conditional language, describing situations in which an author may want to restart a service only when its configuration changes [6]. The practice benefits roles that manage services or configuration files, while roles that make no such changes have no use for it. The benefit depends on the context of the role, so the attribute is classified as recommended.

[4] Ansible Project, "Ansible Tips and Tricks – Ansible Documentation," section "Keep vaulted variables safely visible," 2024, https://docs.ansible.com/ansible/latest/tips_tricks/ansible_tips_tricks.html.

[5] Red Hat Community of Practice,"Automation Good Practices," section "Never store secrets in Git repositories," 2024, https://redhat-cop.github.io/automation-good-practices/.

[6] Ansible Project, "Handlers: Running Operations on Change – Ansible Documentation," introductory paragraph, 2026,https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_handlers.html.

- ***Password (discouraged)***: The Password attribute detects hard-coded credentials in role content, a practice catalogued as CWE-798 (Use of Hard-Coded Credentials) [7]. The same guidance that requires encrypted storage of sensitive values forbids storing them in plain text, so the attribute measures the violation of an obligation rather than a weaker form of advice. The attribute is therefore classified as discouraged and penalised through the negative flag described above.

The values 1.0 and 0.5 encode the order of the three classes rather than measured quantities, because the documentation language establishes that a mandatory practice outranks a recommended one without stating by how much. The absolute scale is immaterial. The category score in Section 3.3.3 divides by W_{total}, so multiplying every weight by the same constant leaves every score unchanged, and configurations such as $\{2.0, 1.0\}$ or $\{5.0, 2.5\}$ produce results identical to $\{1.0, 0.5\}$. The only free parameter is therefore the ratio between the two positive classes. We set it to 0.5 rather than a smaller or larger value because it is the simplest ratio that keeps both classes visible in the score, because it reuses the point scale $\{0.5, 1.0\}$ applied to categories in Section 4.2.2, and because no ground truth for quality exists against which a ratio could be fitted. Discouraged attributes contribute with unit weight after inversion because they measure the violation of an obligation, which the rule treats as equal in strength to meeting one. The ratio is a documented default that organisations may change through the policy mechanism.

At the category level, the best-practice policy assigns equal weights to each of the nine categories ($w_c^{cat} = 1.0$), and the weighted attribute levels dictate the quality differences between solutions, thereby preventing any category biases from influencing the weighted average quality score. The best-practice policy also demonstrates that the framework can do more than assigning varying levels of importance to attributes. Through the classification of attributes as

[7] MITRE, "About CWE-798: Use of Hard-coded Credentials," 2018, https://cwe.mitre.org/data/definitions/798.html.

mandatory, recommended, or discouraged, the framework can reward repositories that follow community standards and penalise those that exhibit anti-patterns.

In sections 4.6 and 4.7 we present an empirical analysis comparing the equal-weight and best-practice policies. This comparison evaluates whether the framework can reflect community standards when different weight configurations are applied to the same set of repositories. By examining how scores change between the two policies, we demonstrate the effect of attribute-level weight differentiation on quality assessment outcomes.

4.4 Category-Level Policy Configurations

Organisations utilising infrastructure automation have unique quality priorities that are based on their operational environment and level of maturity within their development processes [12]. While technical best practices are encoded at the attribute-level, organisational intentions and development philosophies are encoded at the category-level. This section discusses how organisations can convert their quality strategic priorities about quality into quantitative category weights, which will enable the framework to evaluate the repository against the organisation's intentions.

Through a review of organisational case studies, we identified eight development philosophies that represent distinct approaches to infrastructure quality trade-offs. Each philosophy was derived from documented practices of organisations operating under different constraints, as referenced in the

subsequent subsections [64, 65, 66][8][9][10][11][12]. We encode these philosophies as category weight configurations where $w_c^{cat} \in \{0, 0.5, 1.0\}$ for each category c. The weight values encode organisational priority levels: $w_c^{cat} = 0$ indicates dimensions ignored or deemed irrelevant to organisational goals, $w_c^{cat} = 0.5$ represents recommended quality aspects considered beneficial but not essential, and $w_c^{cat} = 1.0$ designates mandatory dimensions central to the organisation's development philosophy. Each configuration creates distinct quality profiles that align with organisational objectives. The following sections provide details about each philosophy along with its relevant use cases.

4.4.1 Policy Configurations by Development Philosophy

We have identified eight philosophies that can be grouped into three categories of growth which range from small to large organisations. Organisations in early growth phases prioritise development speed to bring products to market. Organisations that have achieved product-market fit shift their focus toward excellence in specific quality dimensions that are relevant to their operations. Organisations at the enterprise level possess the resources to pursue quality across all dimensions to meet their business requirements. Our framework allows organisations to select a philosophy from the category that corresponds to their growth stage and business strategy. Our framework also supports transitions between categories, as organisations may adopt different philosophies or combine elements from more than one category as their operational requirements evolve. The three categories, Speed-Focused Development, Spe-

[8]Eric Ries, "Minimum Viable Product: A Guide," August 2009, http://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html.

[9]Todd Hoff, "The WhatsApp Architecture Facebook Bought for $19 Billion," February 2014, http://highscalability.com/blog/2014/2/26/the-whatsapp-architecture-facebook-bought-for-19-billion.html.

[10]Microsoft Corporation, "Zero Trust Architecture," 2024, https://learn.microsoft.com/en-us/security/zero-trust/zero-trust-overview.

[11]Cory Bennett and Ariel Tseitlin, "The Netflix Simian Army," July 2011, https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116.

[12]IBM Corporation, "Technical Debt: Managing Long-term Software Quality," 2024, https://www.ibm.com/think/topics/technical-debt.

cialised Excellence, and Comprehensive Quality, are presented in the following subsections along with their constituent philosophies. Figure 4.1 provides an overview of this philosophy organisation.



Figure 4.1: Organisation of Eight Development Philosophies Across Three Growth Categories

4.4.1.1 Speed-Focused Development

In this category, we identified three philosophies that adopt a "delivery first" approach as part of the organisational culture during the growth phase and/or market capture phase (e.g., the period when an organisation grows quickly in terms of customer acquisition).

Move Fast and Break Things [64] – This philosophy was inspired by the pre-2014 Facebook developers [64]. It represents an organisation's desire to deliver features as quickly as possible, without regard to the technical quality of the features being delivered, including documentation, coding practices, and maintainability. This enables the organisation to gain quick feedback from their customers and users, allowing them to adapt to changing needs based on the information they receive.

Minimum Viable Product (MVP) [13] – An organisation in its startup phase using this philosophy will adopt a minimal viable product strategy to validate whether there is demand for a product or service in the market. In this case, the organisation is balancing the need to deploy the product or service

[13]Eric Ries, "Minimum Viable Product: A Guide," August 2009, http://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html.

quickly with a lower level of quality than would be desired at a later stage. This philosophy recognises that, although deploying the product or service quickly is still a key element, the organisation cannot abandon all of its basic security measures and organisational structure. This philosophy is best suited to organisations with limited resources that need to determine if there is a market for their product or service before developing a high-quality offering.

Lean Infrastructure [14] – The third philosophy in this category is Lean Infrastructure. This philosophy draws from the architectural approach adopted by WhatsApp, where a small engineering team maintained infrastructure for hundreds of millions of users by eliminating all components that did not directly contribute to core functionality[15]. Each component must provide value to the organisation. Otherwise, it is rejected as an organisational expense that does not provide proportional operational benefits. Like the previous philosophies, this one is also suited to organisations with limited resources and budgets that must find ways to achieve efficiency in order to survive.

4.4.1.2 Specialised Excellence

In this category, we identified two philosophies that demonstrate an organisation's emphasis on particular quality dimensions based on the context of their operations, where certain aspects are particularly critical to the operational needs of the organisation.

Documentation-Driven Development [65] – Organisations such as Stripe and HashiCorp place an emphasis on documentation quality for knowledge transfer purposes and API clarity over implementation complexity. The Documentation Driven Development philosophy is apparent in the context of tool specific companies, where the quality of documentation has a direct relationship to customer success. Clarity is valued over technical sophistication so

[14]Todd Hoff, "The WhatsApp Architecture Facebook Bought for $19 Billion," February 2014, http://highscalability.com/blog/2014/2/26/the-whatsapp-architecture-facebook-bought-for-19-billion.html.

[15]Todd Hoff, "The WhatsApp Architecture Facebook Bought for $19 Billion," February 2014, http://highscalability.com/blog/2014/2/26/the-whatsapp-architecture-facebook-bought-for-19-billion.html.

that infrastructure code can be understood and maintained by different teams.

Security-First Development [16] – This philosophy is informed by both the DevSecOps methodology, which integrates security practices into every stage of the development lifecycle, and the Zero-Trust security model, which assumes no implicit trust within or outside the network boundary. Organisations implement this philosophy when they operate in high-risk environments or handle sensitive information. These organisations accept a slower rate of delivering new features and a reduced ability to integrate with external systems in order to achieve comprehensive security controls and error handling mechanisms across their infrastructure.

4.4.1.3 Comprehensive Quality

In this category, we identified three philosophies that represent quality across multiple dimensions. These are typical of large organisations that have sufficient resources to achieve comprehensive quality.

Enterprise Fortress [66] – Large regulated industries such as banking, healthcare, and defence tend to use this philosophy primarily to meet their regulatory obligations and to ensure code lifecycles spanning decades. Enterprise Fortress is therefore a philosophy of control, auditability, and reliability rather than innovation. It is also appropriate when the cost of developing a system is less than the cost of failure.

Chaos Engineering [17] – Organisations that aim to achieve resilience through Chaos Engineering deliberately introduce failures into their systems to build resilience. Chaos Engineering was first developed at Netflix and has been implemented by many of the largest cloud providers. In these environments, it is assumed that all systems will eventually fail, so they are designed to degrade gracefully and recover quickly from failure.

[16] Microsoft Corporation, "Zero Trust Architecture," 2024, https://learn.microsoft.com/en-us/security/zero-trust/zero-trust-overview.

[17] Cory Bennett and Ariel Tseitlin, "The Netflix Simian Army," July 2011, https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116.

Sustainable Engineering [18] – Organisations that plan to operate their software or systems over extended periods are unwilling to make quality trade-offs on any dimension. Organisations such as Google, Microsoft, and IBM apply Sustainable Engineering to the design of their core infrastructures and accept greater upfront costs to provide the long-term maintainability and evolutionary capacity required of them.

4.4.2 Applying Category-Level Policies

To apply organisation-wide policies at the category level, the category weights (w_c^{cat}) defined in Section 4.2.2 are assigned values according to the selected development philosophy. These weights are then applied to the raw quality scores to compute an alignment score, which measures the degree to which a repository conforms to the selected philosophy. The alignment score ($S^p_{alignment}(i)$) is calculated for each repository i as follows:

$$S^p_{alignment}(i) = \frac{\sum_{c \in C_{active}} w_c^{cat,p} \cdot S_{i,c}}{\sum_{c \in C_{active}} w_c^{cat,p}} \times 100$$

where $w_c^{cat,p}$ indicates the category (c) weight according to philosophy (p). With this calculation, it is possible to compare the quality of repositories within the same organisational context.

For example, consider two repositories A and B. Repository A achieves 85% alignment with "Move Fast and Break Things," while Repository B achieves 62%. In this case, Repository A demonstrates higher conformance with speed-focused development values. However, Repository A achieves only 20% alignment with "Enterprise Fortress," while Repository B achieves 82%. From this comparison, Repository B is better suited to an environment where regulatory compliance is a priority.

In addition to identifying who follows the organisational philosophies, the alignment metric allows organisations to develop initiatives to improve the

[18] IBM Corporation, "Technical Debt: Managing Long-term Software Quality," 2024, https://www.ibm.com/think/topics/technical-debt.

quality of repositories where alignment is low, and to verify that all repositories in the organisation's portfolio meet similar quality standards. The dual-level policy weighting system that combines category-level philosophy weights with attribute-level weights allows organisations to achieve technical excellence while adhering to organisational constraints. The subsequent chapters present examples of how organisations can use the category-level policy weights in practice to showcase the framework's ability to quantify quality assessments.

4.4.3 Summary of Philosophy-Based Configurations

This section summarises how each of the eight development philosophies has been translated into an organisation-specific configuration of the category weight hierarchy, as shown in Table 4.1. Organisations can use these configurations as references when choosing which development philosophy will best support their needs and strategic goals. The weights assigned to each philosophy represent our interpretation and translation of each philosophy's organisational principles into numerical values that encode their relative priorities as they apply to the development process. Each philosophy is assigned weights based on the analyses of the core tenets and operational constraints inherent to that philosophy.

Table 4.1: Category Weights for Different Development Philosophy Approaches

Category	Move Fast	MVP	Lean Infra	Doc-Driven	Security First	Enterprise Fortress	Chaos Engg	Sustainable Engg
Metadata	0.0	0.0	0.0	1.0	1.0	1.0	1.0	1.0
Code Structure	0.0	0.5	0.5	1.0	0.5	1.0	0.5	1.0
Code Sophistication	0.5	0.0	0.0	0.0	0.5	0.5	1.0	0.5
Code Maintainability	0.0	0.5	0.5	1.0	0.5	1.0	0.5	1.0
Code Security	0.5	0.5	1.0	0.5	1.0	1.0	1.0	1.0
Functionality & Purpose	1.0	1.0	1.0	1.0	0.5	1.0	1.0	1.0
Error Handling	0.0	0.5	0.5	1.0	1.0	1.0	1.0	1.0
Code Automation	1.0	1.0	1.0	0.5	1.0	0.5	1.0	0.5
Code Integration	1.0	0.5	0.5	0.5	0.5	0.5	0.5	1.0
Total Weight	**4.0**	**4.5**	**5.0**	**6.5**	**6.5**	**7.5**	**7.5**	**8.0**

Legend: 0.0 = Not focused (ignored category), 0.5 = Recommended (beneficial but not essential), 1.0 = Mandatory focus

The weights assigned to each philosophy have been designed to capture each philosophy's underlying principles and operational constraints. Philosophies focused on speed assign full weight (i.e., 1.0) to categories that enable rapid deployment. For example, "Move Fast and Break Things" assigns full weight to Functionality & Purpose, Code Automation, and Code Integration because these three categories directly enable the rapid delivery of new features, while assigning zero weight to documentation and structural considerations. In contrast, "Startup Minimum Viable Product" assigns partial weights (i.e., 0.5) to both structure and maintainability because, although speed remains the priority, organisations still need sufficient organisation and maintainability to sustain rapid deployment. "Lean Infrastructure" eliminates all non-essential categories, assigning full weight only to Code Security, Functionality & Purpose, and Code Automation, while assigning zero weight to metadata, sophistication, and integration components, which are deemed non-essential for delivering value to users.

Philosophies focused on specialised areas of expertise illustrate selective emphasis through different weight patterns. "Documentation-Driven Development" assigns zero weight to Code Sophistication, as simpler code facilitates comprehension and documentation. It emphasises structural clarity and maintainability by assigning full weights to the structural categories, while reducing the weights of Code Automation and Code Integration to 0.5. Although these categories are important for the success of a project, higher weights would risk adding complexity that hinders user understanding. In contrast, "Security-First Development" inverts this pattern by assigning full weight to the security categories while reducing the weights of Functionality & Purpose and Code Integration to 0.5. The rationale being that regardless of what functionality a feature provides, it is never acceptable to introduce any vulnerability to potential attackers or to introduce any external dependency that may pose a risk to the system.

Philosophies that emphasise comprehensive quality exhibit a balanced dis-

tribution of weight among the categories. "Enterprise Fortress" maximises control and reliability by assigning full weights to seven categories, while reducing the weights of Code Sophistication, Code Automation, and Code Integration to 0.5 in order to maintain human oversight over automated processes. "Chaos Engineering" maximises resiliency by assigning full weights to Code Sophistication and Code Automation to create self-healing systems, while assigning a 0.5 weight to Code Integration to promote loose coupling that reduces the likelihood of cascading failures. Finally, "Sustainable Engineering" exhibits the most balanced weighting assignment, assigning full weights to seven categories while maintaining the weights of Code Sophistication and Code Automation at 0.5 to avoid over-engineering or over-automating, which would limit long-term maintainability.

So far, this chapter has established the theoretical foundation for policy-based quality assessment within the framework. This chapter introduced the two-level policy mechanism, consisting of attribute-level weights that encode technical best practices and category-level weights that encode organisational development philosophies. At the attribute level, we defined two configurations, an equal-weight baseline and a best-practice configuration derived from Ansible community standards. At the category level, eight development philosophies were identified across three organisational growth categories, each producing a distinct quality profile through its category weight configuration.

An important consideration when applying category-level policies is that raw category scores cannot be compared directly across categories. Each category contains a different number of attributes with different distributions, so a score of 0.7 in Code Security does not carry the same meaning as a score of 0.7 in Code Maintainability. For example, if Repository A scores 0.8 in Code Security but 0.4 in Code Maintainability, and Repository B scores 0.5 in both, a direct comparison of raw scores would not indicate which repository better aligns with a given development philosophy. To address this, we use the alignment score defined in Section 4.2.2, which applies the philosophy's cate-

gory weights to produce a single weighted percentage that reflects how well a repository's quality profile corresponds to the priorities of a given philosophy. This alignment score enables meaningful comparison of repositories against the different development philosophies.

The following sections describe how these policy configurations are evaluated empirically. Our evaluation begins with a description of the dataset used for the analysis, including the platform selection, data collection methodology, and attribute extraction process. The chapter then presents the results of applying both attribute-level and category-level policies to the dataset, examining how different configurations affect quality assessment outcomes across the repository population.

4.5 Evaluation Dataset

To validate the policy mechanism of the framework, this section describes how the evaluation dataset was created, selected, and collected, and how the data were used to assess the configuration policies at both attribute and category levels.

4.5.1 Platform Selection and Dataset Characteristics

The empirical evaluation requires a dataset drawn from a single IaC platform to ensure consistency in quality assessment. This requirement reflects an evaluation decision rather than a property of the framework, and Section 4.8.3 discusses how the framework transfers to other platforms. We selected Ansible as the primary platform for the remainder of this thesis for two reasons. First, Ansible is one of the most adopted configuration management tools in the IaC domain (Section 2.1.1), providing a large population of repositories for analysis. Second, Ansible has an established ecosystem with documented community standards and best practices, which provides the foundation for deriving the best-practice weight configurations described earlier in this chap-

ter.

Within the Ansible ecosystem, we use Ansible Galaxy as the principal source of data. Ansible Galaxy is a public repository focused on Ansible content, offering several benefits for empirical analysis. Ansible Galaxy enforces metadata field values across all repositories that comply with its standards for structuring repositories, which facilitates the extraction of attribute values. Ansible Galaxy is an Ansible-only platform, so no technology filtering is required when analysing the platform's content. Ansible Galaxy also includes a range of infrastructure automation scenarios and software technologies, from simple service configurations to multi-tiered deployment scenarios, providing sufficient variability to test the framework.

At the time of collection, the dataset consisted of 26,986 repositories with complete metadata records out of a total of 27,791 repositories available on Ansible Galaxy, representing 97.1% of the repository content on Ansible Galaxy. Our inclusion criteria required that repositories have complete metadata records, including repository statistics (downloads and stars), temporal information (repository creation date and last update date), and structural indicators (presence of documentation and whether the platform supports the repository). The 805 excluded repositories lacked one or more required metadata fields, preventing complete assessment in the Metadata category. The characteristics of the repositories included in the dataset are as follows:

- **Temporal Coverage**: The creation dates of the repositories ranged from 2014 to 2023, capturing the evolution of practices over nearly nine years.

- **Size Distribution**: The number of tasks and roles within each repository ranged from a single task or role to complex multi-component architectures.

- **Community Engagement**: The download counts for the repositories ranged from zero to 26,193,507, indicating the degree of community adop-

tion for each repository.

- **Maintenance Activity**: The update frequency for each repository ranged from abandoned repositories to actively maintained projects.
- **Domain Applications**: The repositories cover a wide range of infrastructure automation applications, including web services, databases, security hardening, monitoring, and application deployment.

4.5.2 Data Collection Methodology

The methodology for collecting data in this study used an automated approach to extract information from Ansible Galaxy's public API, along with a mechanism to collect the source code. The collection process followed four sequential stages, where each stage depended on the successful completion of the preceding stage.

1. **Metadata Extraction**: The process began by querying Ansible Galaxy's API endpoints to retrieve repository-level information, including statistics, version history, and supported platforms.
2. **Source Code Collection**: Using the metadata obtained in the previous stage, the `ansible-galaxy` command-line tool downloaded the entire contents of each role identified through the API.
3. **Validation Checks**: Each downloaded repository was then verified to confirm that the source code had been successfully retrieved. Repositories that contained corrupted or incomplete source code were excluded from further processing.
4. **Attribute Parsing**: For all repositories that passed validation, a custom Python analyser performed static analysis on the YAML files and role structures to extract code-level metrics.

The extraction process captured snapshots of each repository's state at the time of collection. Additionally, the methodology included capturing historical

metadata (i.e., creation date) to enable longitudinal analysis of how the quality of each repository evolved over time.

4.5.3 Attribute Extraction and Mapping

Table 4.2 presents the same 96 attributes as Table 3.9, with the addition of colour coding to indicate the best-practice weight classification assigned to each attribute. Green denotes mandatory attributes ($w_{c,j} = 1.0$), blue denotes recommended attributes ($w_{c,j} = 0.5$), and red denotes discouraged attributes that are penalised during quality assessment. These classifications record the outcome of applying the decision rule described in Section 4.3.2 to the official Ansible documentation and community standards for each of the 96 attributes

.

The attribute extraction process employed four complementary techniques depending on the nature of each attribute. Metadata attributes such as download count, stars, forks, and version history were obtained directly from the Ansible Galaxy API responses collected during the metadata extraction stage, while additional metadata attributes including dependencies, supported platforms, and minimum Ansible version were parsed from each repository's `meta.yml` file. Structural attributes (directory count, file count, lines of code, blank lines) were computed through file system traversal of each downloaded repository. Code-level attributes (loops, conditions, vault usage, error handling constructs, module usage) were extracted through regular expression pattern matching applied to YAML and Jinja2 template files within each repository. Functionality and syntax validation attributes were computed using `ansible-lint`, which performed static analysis on each repository to identify syntax errors, deprecated module usage, and deviations from Ansible coding conventions.[19]

[19] "IaC Quality Assessment Framework: Extraction Pipeline, Scoring Algorithms, and Analysis Scripts," 2025, available at https://figshare.com/s/1f8d98b7073447a7f63d.

Table 4.2: Mapping of Code Attributes to Proposed Code Quality Framework Categories

IaC Code Quality Category	Code Attributes
Metadata	Download count, Tag, Total versions, Dependencies, Supported platform count, Stars, Forks, version release time, Github status, Minimum Ansible versions, Average update time, Open issues
Code Structure	Count of lines in YAML/YML files, Lines of code in YAML/YML files, Task count, Count of directories, Variables, Files count, Lines in source code, Blank spaces between words, Average play size, Average task size, Unique names, Names with variables,Length of task, Template files, Line blanks
Code Sophistication	Entropy, Loops, Conditions, Decisions, Include, Ensure, Keys, All modules, Mathematical operations, Regex, Lookups
Code Maintainability	README word count, Comments, User interactions
Code Security	SELinux, Firewalld, Vault, stat, File modes, No Log, apt_key, SSH keys, Password, Suspicious comments, Deprecated keywords, Deprecated modules
Functionality & Purpose	README count, License file presence, Syntax Check
Error Handling	failed_when, changed_when, *Rescue**, *Always**, *Block**,Debug, any_errors_fatal settings, max_fail_percentage,*Retry**, *Until**, *Delay**,ignore_errors
Code Automation	Become, vars, check_mode, Paths, Plays, Roles,*URI*$^{\curlywedge}$, *URL*$^{\curlywedge}$,Fact modules, Filters, environment, local_action, Fetch modules, Handlers, Tags,Commands
Code Integration	wait_for, Git, Import playbooks, Import *roles*$^{\uplus}$, Include *roles*$^{\uplus}$, Import *tasks*$^{\bowtie}$, Include *tasks*$^{\bowtie}$,Include vars, Rsync, win_service, add_host

Legend: ■ Attributes penalised by best practices - degrades quality
■ Recommended attributes ($w_{c,j} = 0.5$) - beneficial but not mandatory
■ Mandatory attributes ($w_{c,j} = 1$) - required by Ansible best practices
Symbols ($\star$, $*$, $\curlywedge$, $\uplus$, $\bowtie$) indicate grouped attributes to prevent double counting

4.6 Attribute-Level Policy Evaluation and Results

To evaluate how effectively the framework assesses quality, an empirical evaluation is performed on 26,986 Ansible Galaxy repositories. This section presents the results of the assessment using the best-practice policy, which represents typical deployment use cases, alongside comparative results using an equal-weight policy to demonstrate how different policies affect assessment outcomes.

To ensure that quality score comparisons across repositories of different sizes are fairly represented, all code-based attributes are normalised per 100 lines of code. This normalisation prevents larger codebases from achieving

higher quality scores solely because they contain more attributes due to their size. There are two exceptions to this normalisation process. First, attributes contained in the Metadata category are not normalised, as their values are independent of code volume. Second, size-invariant metrics (e.g., total directories, total files) are also exempt, since their absolute values convey important quality information about the repository. For example, a repository with 1,000 lines of code that includes extensive error handling would receive an equivalent quality score to a repository with 10,000 lines of code with proportionate levels of error handling. Conversely, whether a repository contains sophisticated security features or error handling mechanisms is evaluated relative to the amount of code present. This normalisation method ensures that each repository is assessed equitably while maintaining the ability to interpret quality metrics relative to code size.

4.6.1 Best-Practice Policy Analysis

The best-practice policy configuration is based on community standards represented by the weight system as follows: a weight of 1.0 is assigned to mandatory best practices, a weight of 0.5 to recommended practices, and identified anti-patterns which are discouraged during normalisation. The policy applies equal weighting at the category level ($w_c^{cat} = 1.0$ for all nine categories) to allow a full examination of the quality landscape without category bias, so that attribute-level weights alone differentiate quality levels. This configuration represents how organisations would use the framework in practice, reflecting the incorporation of cumulative domain expertise into quality assessments with comprehensive coverage of all quality dimensions.

4.6.1.1 Overall Quality Score Distribution

Figure 4.2 presents the temporal evolution of total quality scores across 6-month periods from 2014 to 2023, combining score trends with repository population dynamics. The repository count (gray bars) grows from 22 in

early 2014 to a peak of 1,839 in mid-2017, before stabilising between 1,059 and 1,785 and declining to 899 by 2023. Average total scores (Orange Line) show steady increases from 4.36 to 4.72 (+8.3%) over the study period. The standard deviation ranges from ± 0.18 to ± 0.29, while the highest scores (Green Triangles) range between 4.70 and 5.28 and the lowest scores (Red Triangles) fluctuate between 3.49 and 3.96.

The concentration of scores within this range reflects the dampening effect of the median-based normalisation described in Section 3.3.2. Because the algorithm centres each attribute distribution around the median and maps all values to [0,1], repositories with typical attribute values receive normalised scores near 0.5. When these normalised values are aggregated across nine categories, each contributing a score in [0,1], the total scores converge toward the midpoint of the [0,9] range. This compression is an expected consequence of relative assessment, where repositories are evaluated against one another rather than against absolute benchmarks.

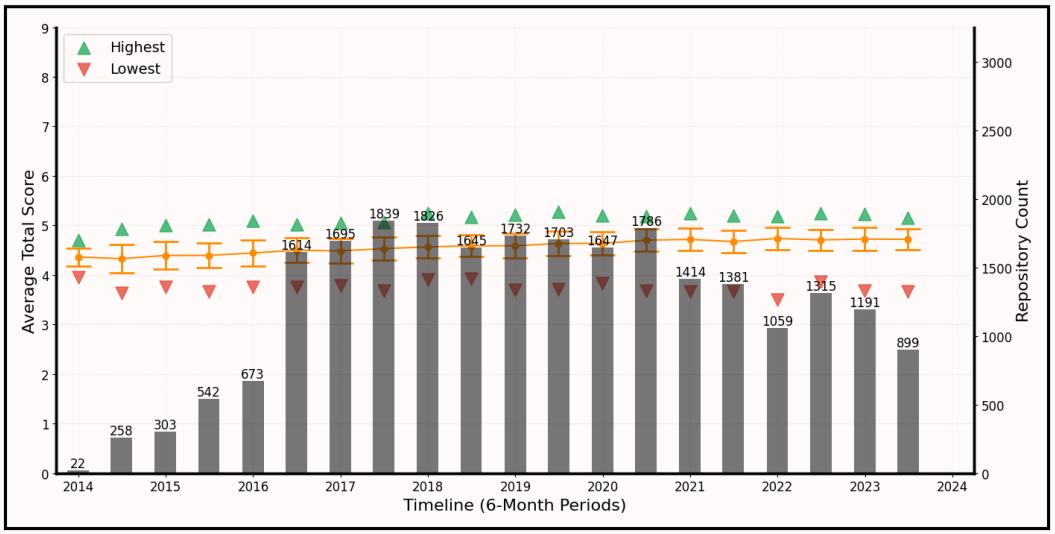


Figure 4.2: Six-Month Aggregate Statistics for Total Quality Scores Across Repository Dataset

To examine how quality evolved over time, we partitioned the temporal data into three phases based on repository population dynamics and conducted separate regression analyses for each. The nascent phase (2014–2015) comprised four 6-month periods with a limited repository population, result-

ing in higher score variability (mean standard deviation of 0.270) compared to later phases (0.238 and 0.229 respectively). The total score slope during this phase was +0.034 points per 6-month period, the steepest of the three phases, though not statistically significant due to the small number of periods ($p = 0.050$, $R^2 = 0.90$, $n = 4$). The growth phase (2016–2019) exhibited both an expanding repository population and the strongest statistically significant quality improvement, with a slope of +0.023 points per 6-month period (p ¡ 0.001, $R^2 = 0.94$, $n = 8$). The mature phase (2019–2023) showed a decelerating slope of +0.012 points per 6-month period ($p = 0.024$, $R^2 = 0.60$, $n = 8$), with scores stabilising around 4.69 and reduced variability across repositories. The progressive reduction in slope across the three phases (+0.034, +0.023, +0.012) demonstrates that quality improvement rates diminished as the platform matured. We observe that these improvements occurred independently of repository population size, as scores continued to rise even as the number of new repositories declined during the mature phase, suggesting that the improvements reflect evolving community practices rather than changes in the volume of contributions.

4.6.1.2 Quality Category Performance Analysis

Next, we discuss the evolution of the nine quality categories within the Ansible Galaxy ecosystem from 2014 through 2023. To ensure comparison across categories, we normalise the category scores to a range of [0,1] while retaining the same temporal bucketing and repository count used in the previous analysis. Based on observed trends, the categories are grouped into three classes: categories with increasing scores (Figure 4.3), categories with stable performance (Figure 4.4), and categories with declining scores (Figure 4.5). Each quality dimension follows a pattern of change that reflects the evolving quality within the ecosystem.

Categories with Increasing Scores: The Metadata category shows increasing scores, with its normalised value increasing from 0.466 to 0.540, a

15.9% gain over the study period. This reflects increased adoption of metadata-based documentation and repository maintenance standards by authors in their submissions to Ansible Galaxy, as contributors provide more complete version histories, dependency statements, and platform compatibility information. The Code Structure category also increased, rising from 0.365 to 0.448 (+22.7%), the largest percentage increase among all categories after Code Maintainability. The Functionality & Purpose category showed a similar trend, with its normalised value increasing from 0.549 to 0.645 (+17.5%) as syntax checking and documentation became common practices among contributors. Consequently, only 14.41% of repositories scored below 0.5 for functionality, indicating that most contributors are adopting a basic level of quality standards. Code Maintainability exhibited the largest percentage increase, rising from 0.325 to 0.424 (+30.5%), though this category retains the lowest absolute scores among all nine categories, with 66.24% of repositories scoring below 0.5, typically due to the prevalence of skeleton or placeholder roles.

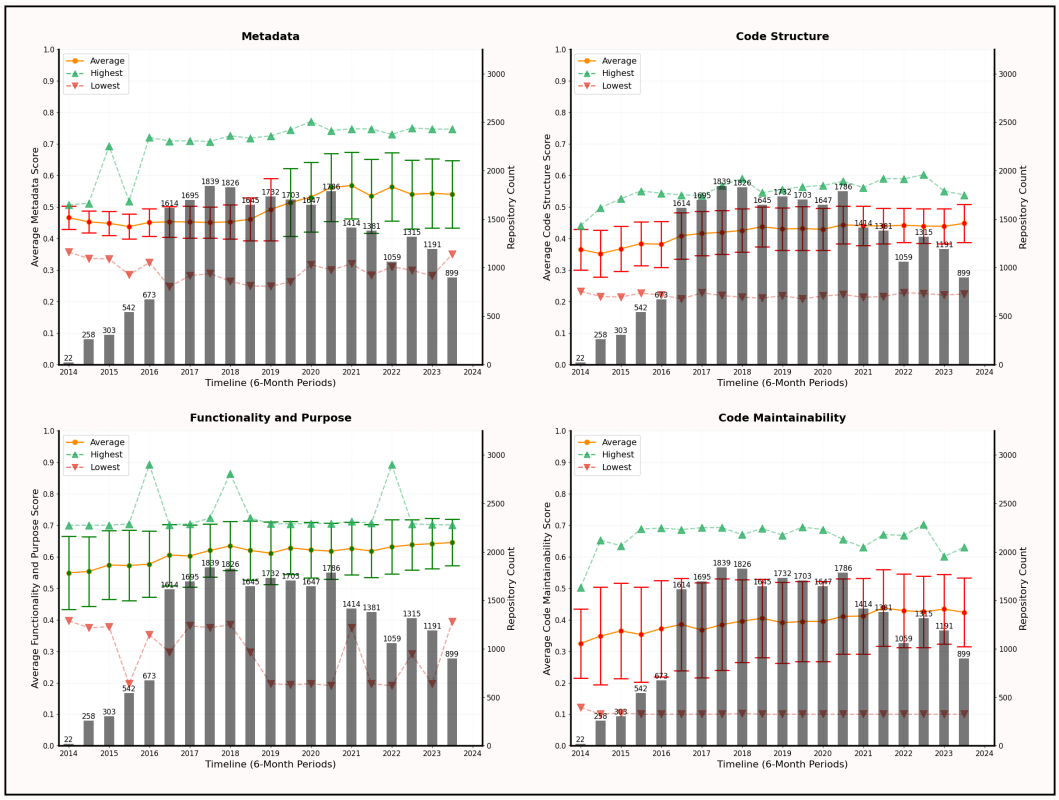


Figure 4.3: Six-Month Aggregate Performance for Categories with Increasing Scores

Stable Performance Categories: Code Security, Error Handling, Code Automation, and Code Integration all exhibit stability in their normalised values throughout the study period. Code Security maintained values in the 0.633 to 0.642 range (-0.6% overall change), indicating that early adopters of Ansible implemented security practices that persisted over time. Error Handling remained at approximately 0.572 to 0.574 throughout the study (+0.3% change), with only 8.26% of repositories scoring below 0.5. Most repositories rely on Ansible's built-in failure mechanisms rather than implementing structured error handling, further characterising most repositories as simple automation scripts rather than production-ready solutions. Code Automation showed limited variation (0.515 to 0.537, +4.3%), suggesting that contributors primarily use Ansible's core automation features to create single-purpose roles with few customisations. Code Integration showed a similar pattern of stability (0.503 to 0.503, +0.0%), with few inter-role dependencies or integration with external systems, demonstrating that most content hosted on Ansible Galaxy consists of independent, single-purpose roles rather than cohesive infrastructure configurations.

Declining Categories: Code Sophistication is the only category that exhibited a decline, with normalised values decreasing from 0.427 to 0.411 (-3.7%) over the study period. Analysis revealed that the primary cause of this decline is the migration of features from third-party modules into Ansible Core. As features migrated into core, repositories required fewer external modules and complex structures, resulting in lower sophistication scores. This reduction may therefore indicate improved maintainability rather than a deterioration in code quality.

Category level analysis reveals that, although the Ansible Galaxy has developed in many categories (metadata, basic functionality, security) to meet the needs of a mature community, the Ansible Galaxy ecosystem may not be developing toward the more complex, enterprise grade solutions that would serve professionals. Furthermore, the continued use of simple, independent

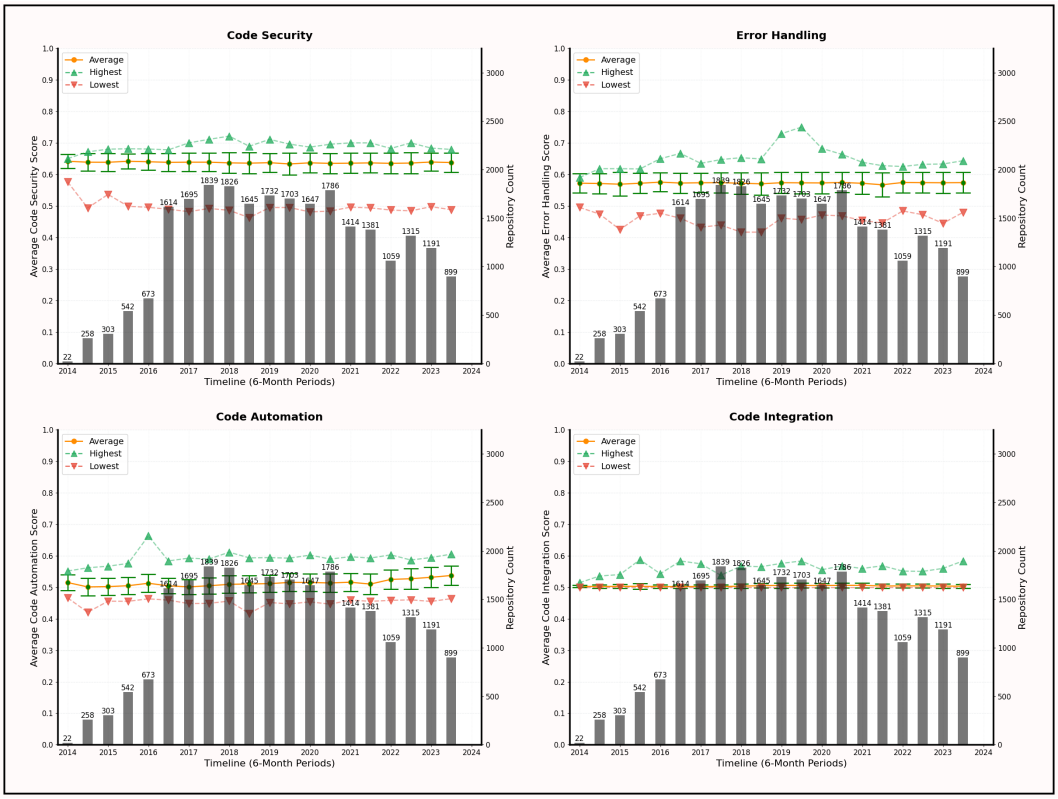


Figure 4.4: Six-Month Aggregate Performance for Categories with Stable Scores

roles implies that Ansible Galaxy is being used as a repository of basic automation components, rather than as a source of sophisticated infrastructure solutions for professionals.

4.6.2 Comparing Equal-Weight and Best-Practice Policies

The application of equal-weight policy to the 26,986 Ansible Galaxy repositories yields similar aggregate results to the best-practice configuration, with mean total scores of 4.542 versus 4.611, a difference of 0.069 points (1.5%) as shown in Figure 4.6. This minimal aggregate variation masks selective category-level differences that validate the policy mechanism's targeted effects. Under equal weights, all attributes contribute uniformly to category scores, revealing natural quality distributions without policy-induced emphasis. The stability of several categories under both policies demonstrates inherent align-

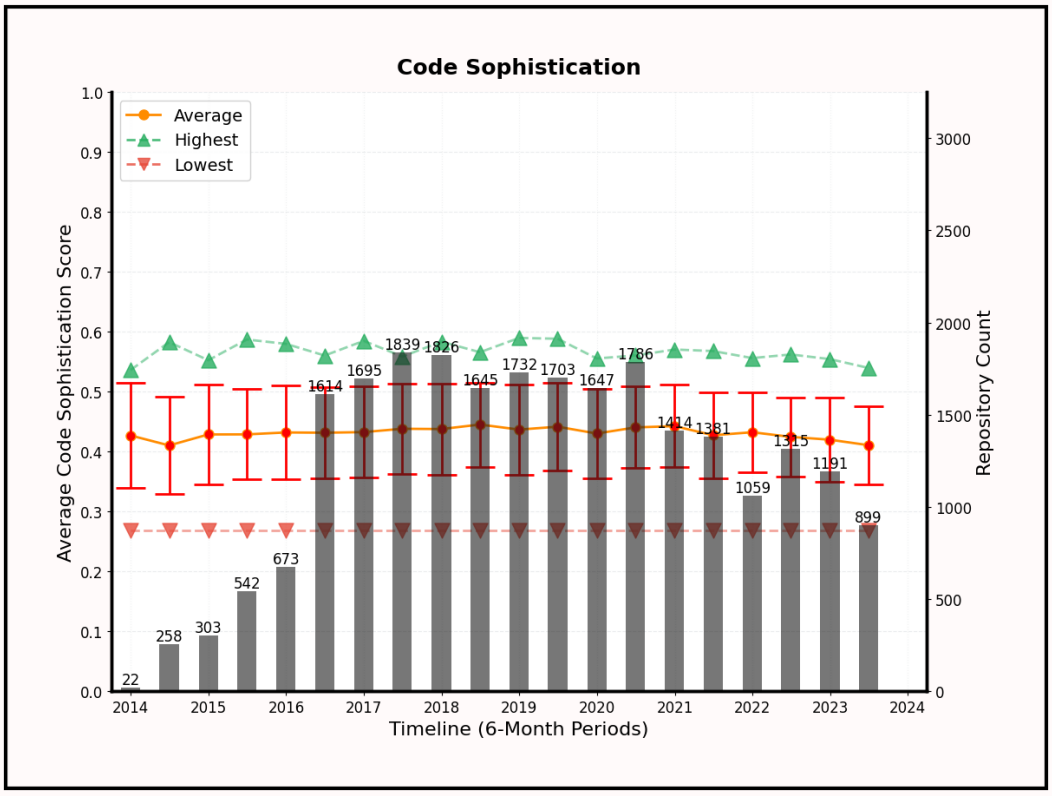


Figure 4.5: Six-Month Aggregate Performance for Categories with Declining Scores

ment: Metadata (0.503), Code Integration (0.503), and Code Security (0.636 vs 0.637) show virtually identical scores, indicating that repositories naturally implement these quality dimensions consistently regardless of weight configuration. This convergence suggests that certain quality aspects represent fundamental requirements in IaC development that emerge organically without policy enforcement.

Category-level variations between policies reveal where best-practice weights create meaningful differentiation. Functionality & Purpose exhibits the largest positive shift under best practices (0.558 to 0.618, +10.7%), driven by mandatory weights on documentation, distribution permissions, and syntax validation that penalise repositories with missing license files. Error Handling improves from 0.555 to 0.573 (+3.2%) as `ignore_errors` is classified as an anti-pattern and discouraged, while structured error management receives higher weight. Conversely, Code Maintainability decreases from 0.417 to 0.401 (-3.8%) and Code Sophistication from 0.444 to 0.431 (-2.9%) under best prac-

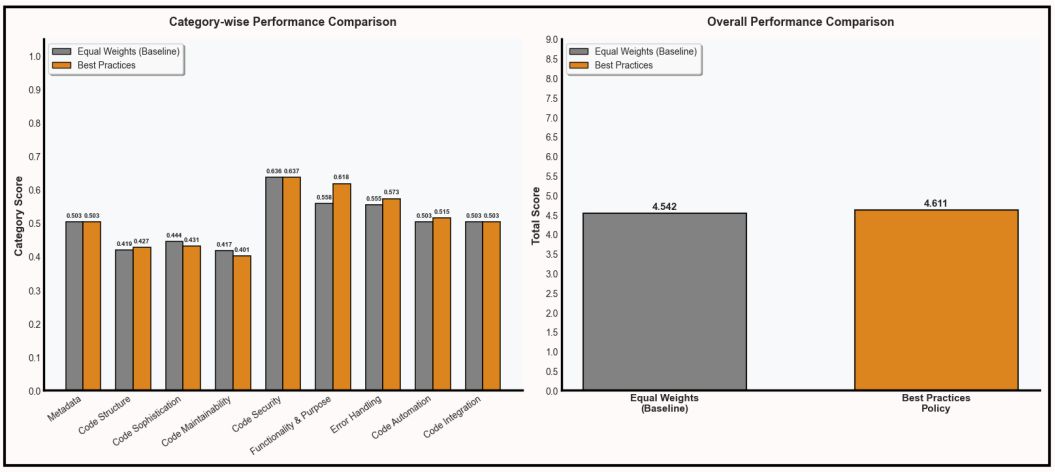


Figure 4.6: Attribute-Level Policy Comparison: Equal-Weight vs Best-Practice Configurations

tices, as the 0.5 weights assigned to recommended attributes in these categories reduce their contribution relative to the equal-weight configuration.

These variations indicate that best-practice weights reinforce the distinctions already present in the data. Repositories that follow mandatory practices receive higher scores, while those that contain anti-patterns receive lower scores. However, the weights do not alter the relative ordering of repositories. A repository that scores higher than another under equal weights continues to score higher under best-practice weights.

These category-level differences require a precise reading. A higher best-practice score in a category does not mean that the repository is more suited to that category, and a lower score does not mean that its quality has decreased. Both configurations score the same repository against different standards. A higher best-practice score means that the category derives its score from practices the community standards classify as mandatory, which the best-practice weights amplify. A lower best-practice score means that the category derives its score from practices classified only as recommended, whose weight is reduced to 0.5, or that the repository contains discouraged anti-patterns, which the configuration penalises. The difference between the two configurations therefore locates where the community standards bite, and characterises the weight configuration rather than the repository. Two reading rules follow. Scores

are comparable across repositories within one configuration, and the relative ordering of repositories is preserved across configurations, so rankings carry meaning under both. Score values are not comparable across configurations, so a repository scoring 5.0 under equal weights and 4.5 under best-practice weights has not lost quality. It has been measured against a stricter standard.

The minimal difference in aggregate scores (1.5%) between the two policies may appear to suggest that the best-practice configuration provides limited value over equal weighting. However, this interpretation does not account for the behaviour of the policy mechanism at the category level. The aggregate score represents an average across all nine categories, which dilutes the effect of category-level differentiation. When examined at the category level, the differences are substantial, reaching 10.7% for Functionality & Purpose and 3.8% for Code Maintainability. These category-level differences demonstrate that the best-practice policy applies the community standards encoded in its weight configuration to distinguish between repositories that follow mandatory practices and those that contain anti-patterns. The small aggregate difference therefore reflects consistency in overall assessment rather than a failure of the policy mechanism. Repositories that follow established community practices achieve similar aggregate scores under both policies because they perform well across all categories. The best-practice policy differentiates repositories at the category level, where the distinction between mandatory, recommended, and discouraged practices is applied. This category-level differentiation is where the policy mechanism provides its primary value, enabling organisations to identify specific areas of strength and weakness within repositories rather than relying on a single aggregate measure.

4.6.3 Repository-Level Quality Evolution Analysis

The preceding analysis examined quality trends at the aggregate level across the entire Ansible Galaxy ecosystem. However, aggregate statistics obscure the development trajectories of repositories, which may exhibit distinct quality

evolution patterns depending on whether they are maintained by organisations or individual contributors. To address this limitation, we next examine how quality evolves at the repository level through longitudinal analysis of complete commit histories, enabling us to observe quality changes as they occur across successive development iterations rather than at a single temporal snapshot.

Longitudinal analysis was selected as the evaluation approach because it captures the incremental quality decisions that developers make throughout a repository's lifecycle. Unlike the analysis presented earlier, which groups repositories into 6-month periods based on their latest version release date, commit-level analysis reveals whether score increases are sustained, reversed, or concentrated in specific development phases. This distinction is important for understanding whether the ecosystem-wide quality trends identified in the previous subsection reflect consistent development practices or are artefacts of aggregation.

To conduct this analysis, we selected six nginx-related repositories as case studies. Nginx was chosen as the target application because it is one of the most widely deployed web servers, ensuring that multiple independent implementations exist on Ansible Galaxy with sufficient commit histories for longitudinal comparison. The six repositories were selected to represent two authorship types: three organisational repositories (nginx.nginx, idealista.nginx_role, ome.nginx) maintained by development teams, and three individual repositories (ansibleguy.infra_nginx, geerlingguy.nginx, jdauphant.nginx) maintained by single developers. This pairing enables direct comparison of quality evolution patterns between organisational and individual development practices. The repositories were further selected to span a range of community adoption levels, from the highly downloaded geerlingguy.nginx (over 13.5 million downloads) to less widely adopted implementations, ensuring that the analysis captures quality trajectories across varying levels of community engagement.

Figure 4.7 tracks quality trajectories across all commits for the six repositories. Organisational repositories, highlighted in shades of blue, achieve av-

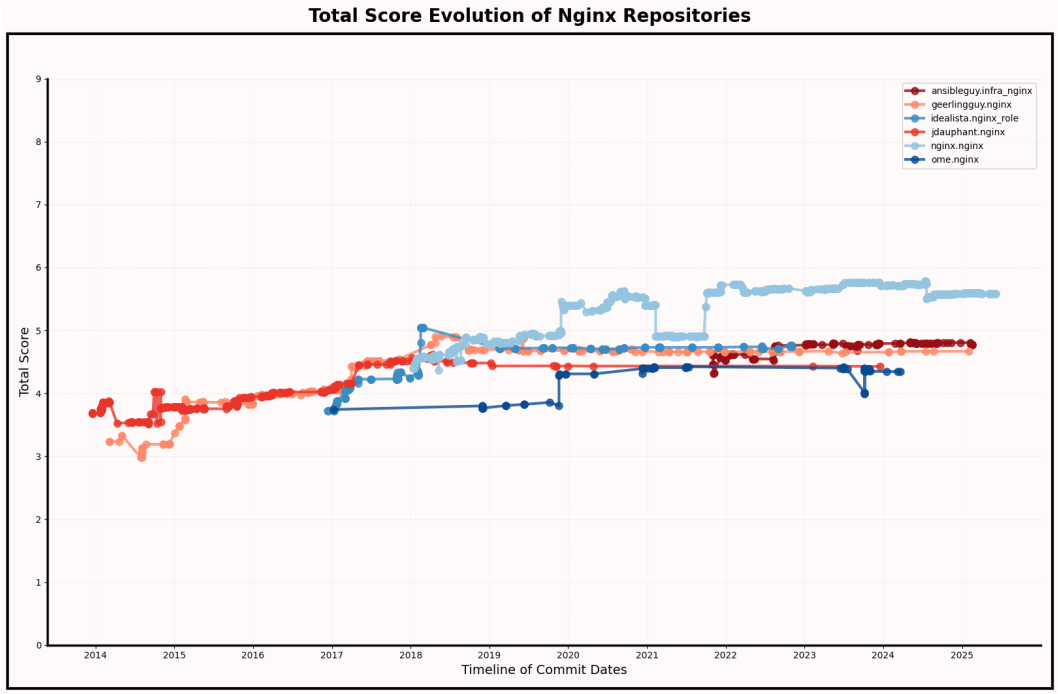


Figure 4.7: Total Quality Score Evolution Across Six Nginx Repositories

erage total scores of 5.26 (nginx.nginx), 4.51 (idealista.nginx_role), and 4.21 (ome.nginx), while individual repositories, highlighted in shades of red, reach average total scores of 4.72 (ansibleguy.infra_nginx), 4.20 (geerlingguy.nginx), and 3.95 (jdauphant.nginx). The official nginx repository (nginx.nginx) demonstrates steady score increases across all categories as shown in Figure 4.8, with Metadata evolving from 0.1 to 0.63 and Code Structure from 0.2 to 0.63 through systematic refactoring cycles. Individual repositories exhibit more volatile evolution patterns, with geerlingguy's repository showing rapid initial development followed by quality plateaus, while ansibleguy's repository demonstrates late-stage score increases particularly in Automation (reaching 0.62, highest among all repositories). All repositories converge on similar Security scores (0.57–0.68) and Integration scores (0.50–0.53), indicating universal adoption of basic security practices and preference for stand-alone role design regardless of authorship type.

The official nginx repository exhibits notable quality fluctuations during 2021–2022, attributed to major architectural refactoring rather than quality degradation. The repository supports multiple nginx variants (basic NGINX,

NGINX Plus, NGINX Agent controller, NGINX Amplify) across diverse operating systems, necessitating periodic restructuring. The fluctuation is a result of the removal of RHEL/CentOS 6 parameters due to end-of-life and the extraction of configuration functionalities into a separate repository, both of which caused temporary score reductions in the Structure and Sophistication categories. Additionally, Functionality & Purpose scores dropped during 2020–2022 due to syntax warnings from using shell attributes instead of Ansible's command module, triggering "command-instead-of-module" warnings despite functional correctness. These architectural decisions demonstrate organisational repositories' willingness to accept temporary reductions in quality scores for long-term maintainability improvements. In contrast, similar patterns are absent in individual repositories that maintain stable but lower quality baselines.

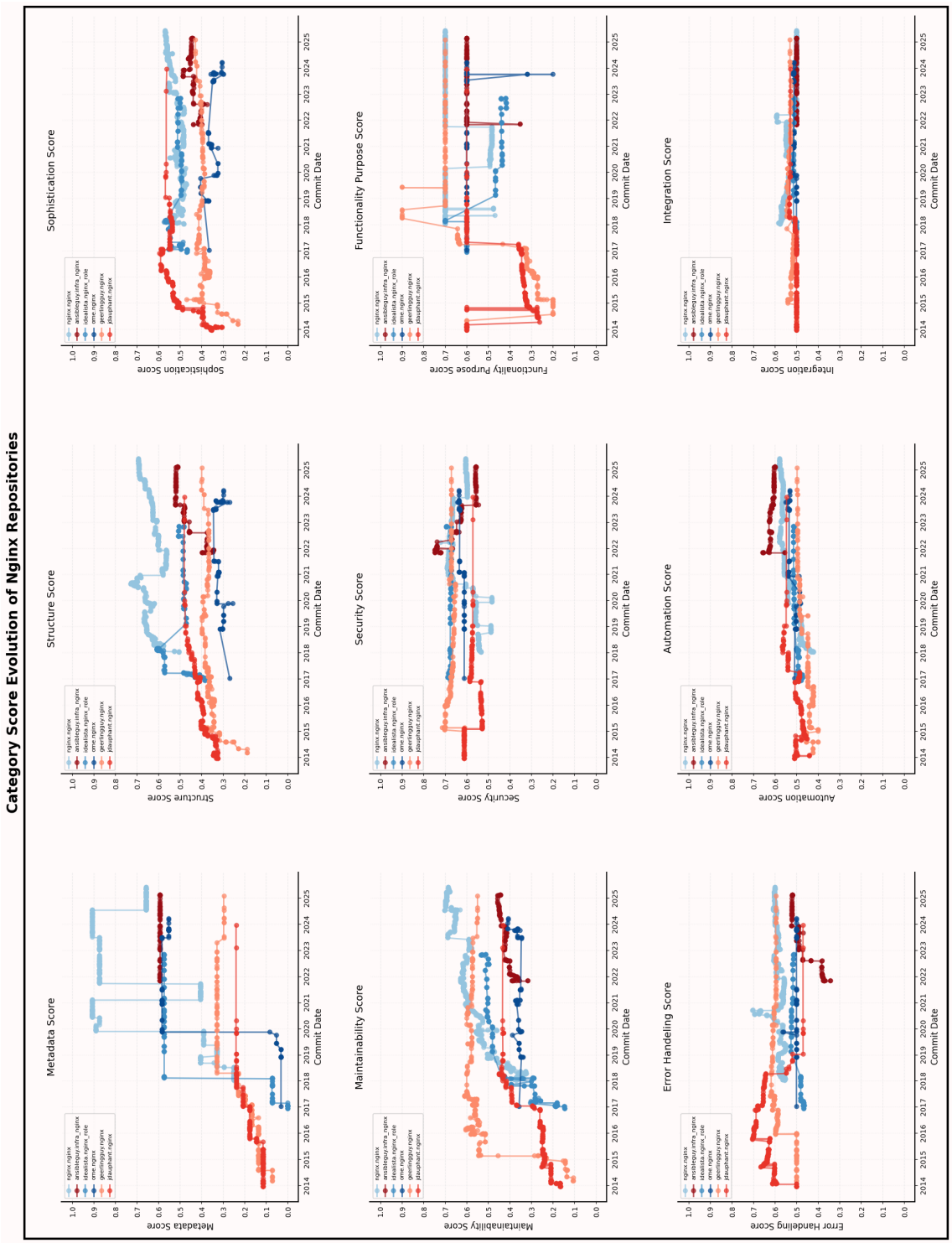


Figure 4.8: Nine Quality Category Score Trajectories for Nginx Repositories

An observation of interest emerges when comparing download counts with quality scores. The `geerlingguy.nginx` repository has over 13.5 million downloads, yet its quality score is lower than that of the official `nginx.nginx` repository, which has 3.8 million downloads. This suggests that quality, as measured by the framework, does not necessarily translate into popularity within the community. Factors such as community trust, the reputation of the author, and the history of the repository may have a greater influence on a user's decision to adopt a particular repository than the overall quality of the code. This finding highlights a broader consideration for practitioners, that quality assessment and adoption decisions are driven by different criteria, and that organisations seeking high-quality repositories cannot rely on download counts as a proxy for code quality.

Organisational repositories demonstrate strong systematic quality characteristics, such as the official nginx repository having 4.5 times better than metadata management, and 2.6 times better than code structure, demonstrating the formal documentation, structured organisation and maintenance cycle of the organisation. Repositories created by individuals can compensate for their lack of systematic quality characteristics by being excellent in a specific area, i.e., jdauphant's repository demonstrates the best error handling (0.63) and ansibleguy's repository demonstrates the most automation features. This also supports the idea that the weights used for best practice, are effective at distinguishing the different ways in which a developer develops: organisational repositories develop quality consistently improving their quality in all areas, however they accept temporary degradation when evolving the architecture, whereas individual repositories focus on select optimisation based on the expertise of the author. Overall, these patterns suggest that the framework captures both, the way in which IaC code evolves (through git commits) and the strategic quality trade-off made by developers working in an enterprise environment.

4.6.3.1 Comparison to Equal-Weight Policy

A comparative study of the quality evolution in six nginx-related repositories was conducted to examine the influence of weighted mechanisms on quality assessment outcomes. As shown in Figure 4.9, the average total score for all versions of the evaluated nginx repositories increased from 4.65 under equal weighting to 4.73 under the best-practice policy, representing a +1.7% change. At the category level, variations in scores were observed across the two configurations. Functionality & Purpose exhibited the largest positive change at +8.5%, followed by Code Automation at +2.3% and Error Handling at +2.0%. Code Security and Code Integration showed smaller positive changes of +1.5% and +0.6% respectively. Code Maintainability experienced a negative change of -1.0% under best practices, primarily as a result of reduced emphasis on `user interaction` attributes. Metadata and Code Structure remained constant across both policy configurations regardless of the repository examined.

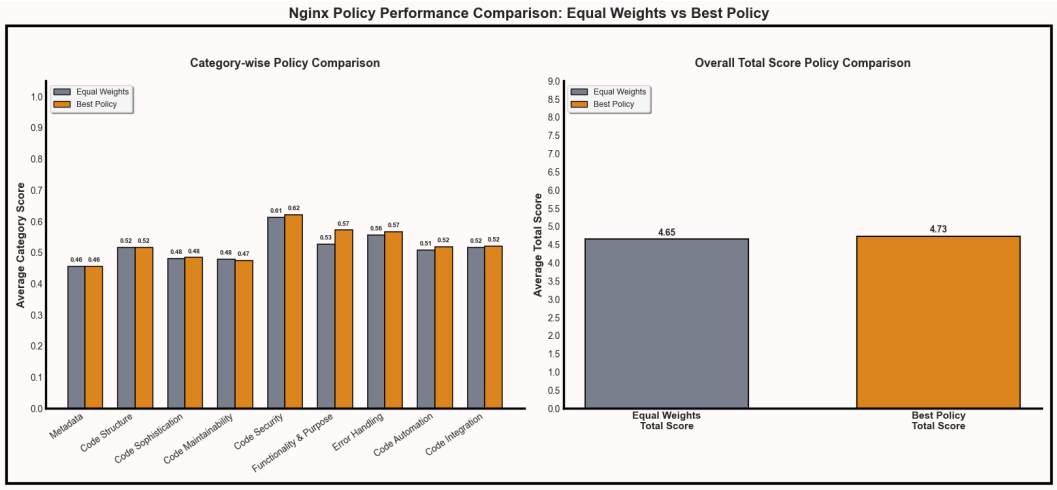


Figure 4.9: Average Quality Category Scores Across Nginx Repositories: Policy Configuration Comparison

Figure 4.10 presents a per-repository comparison of scores under each policy, illustrating how different authorship types exhibit varying degrees of score change between configurations. Organisational repositories show consistent positive changes under best-practice weights. For example, `nginx.nginx` increased by +0.11 points from 5.15 to 5.26, and `ome.nginx` also increased by

+0.11 points from 4.11 to 4.21. In contrast, individual contributor repositories exhibit variable score changes. `geerlingguy.nginx` increased by +0.09, `ansibleguy.infra_nginx` by +0.08, and `jdauphant.nginx` by +0.03. `idealista.nginx_role` received equal scores under both policies, suggesting implementation patterns that align equally with both assessment approaches. The ranking of repository scores remained unchanged after the application of each policy, with `nginx.nginx` at the top (5.26) and `jdauphant.nginx` at the bottom (3.95).

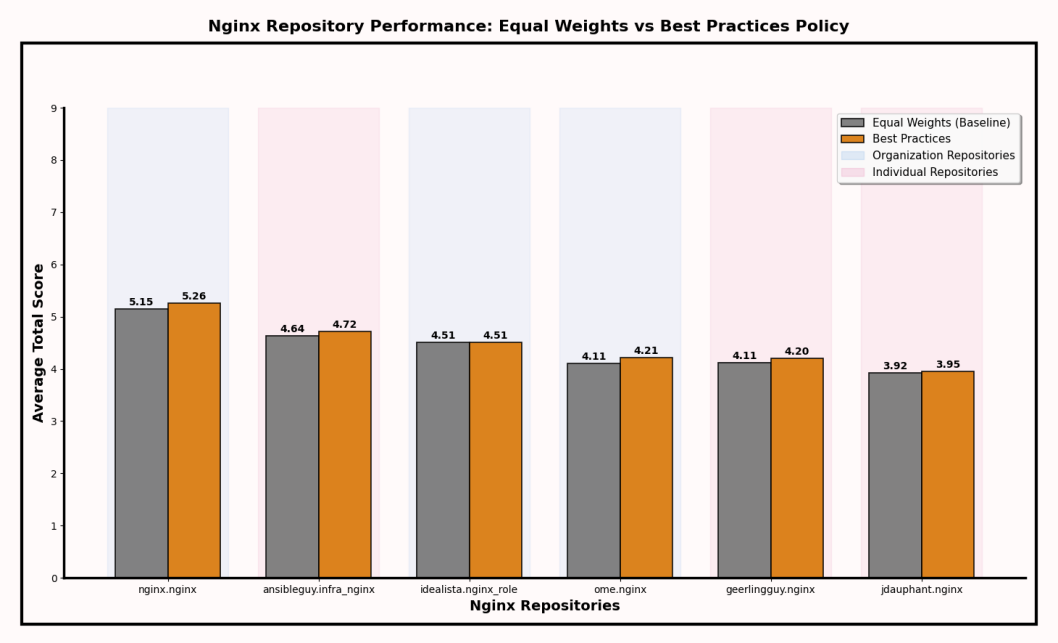


Figure 4.10: Total Quality Scores by Nginx Repository: Equal-Weight vs Best-Practice Policy Comparison

The comparative analysis reveals that best-practice weights produce measurable but selective score differentiation in repository evolution assessment. The +8.5% change in Functionality & Purpose indicates that nginx repositories implementing documentation and proper licensing receive higher scores under community standards than under equal weights. The consistent positive changes in organisational repositories (averaging +0.11) versus the variable changes in individual repositories (ranging from +0.03 to +0.09) suggest that organisational development practices align more closely with best-practice requirements. The preservation of repository ranking under both policies val-

idates that the weight mechanism amplifies existing quality patterns rather than creating arbitrary reordering. This stability confirms that policy configuration reinforces quality signals without disrupting fundamental quality relationships, enabling organisations to apply domain-specific weights while maintaining meaningful repository comparisons throughout their evolution.

4.6.4 Single Repository Analysis Across Policy Configurations

This subsection presents a comparative analysis of specific repositories to demonstrate how project-level quality assessment is affected by policy configuration. While the overall results of the aggregated analysis in the previous section show little variance between policy configurations, individual analyses of repositories reveal the differential effects of policies when applied according to each repository's characteristics.

Two categories of cases are used for this comparative analysis. The first category includes repositories that received higher scores under the best-practice policy than under the equal-weight policy, and the second includes repositories that received lower scores. Both categories demonstrate how the policy enforcement mechanisms selectively affect scores for repositories based on their adherence to community standards.

4.6.4.1 Higher Scores under Best-Practice Policy

The `mrlesmithjr.elkstack` repository receives higher scores when assessed through best-practice weights. Under the equal-weight policy, the repository achieves a total score of 4.687, which increases to 4.909 under the best-practice configuration, representing a +4.7% change. Figure 4.11 illustrates the category-level score variations between policies.

Category-specific analysis reveals higher scores in three dimensions. Code Structure increases from 0.383 to 0.403 (+5.2%), indicating alignment with Ansible Galaxy structural conventions that receive mandatory weights under

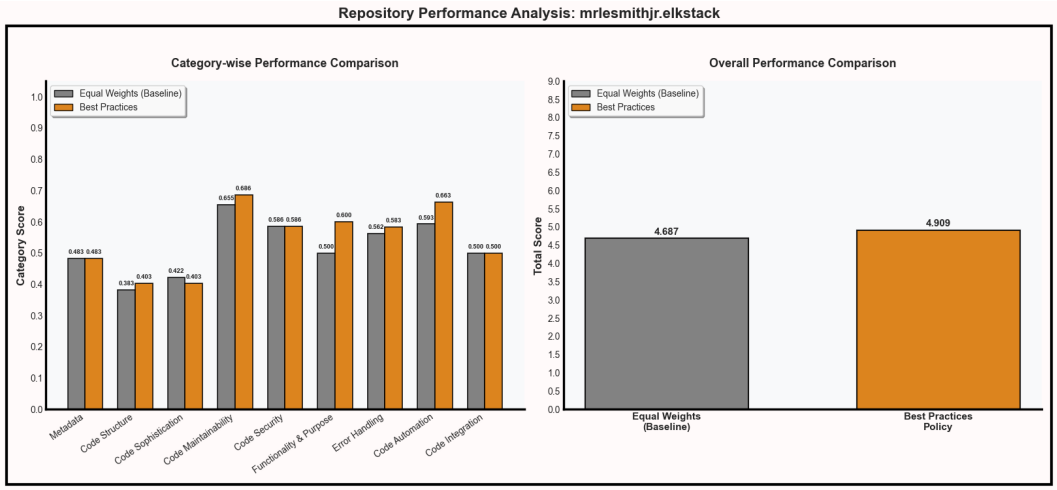


Figure 4.11: Higher Scores Under Best-Practice Policy: `mrlesmithjr.elkstack` Repository

best practices. Code Security improves from 0.655 to 0.686 (+4.7%), reflecting proper implementation of security controls including vault usage and secure file permissions. Functionality & Purpose shows the most substantial gain, rising from 0.5 to 0.6 (+20%), driven by comprehensive documentation and proper licensing that satisfy mandatory requirements.

The repository maintains stable scores in Metadata (0.483), Code Maintainability (0.586), and Code Integration (0.5) across both policies. Minor variations occur in Code Sophistication (-4.5%) and Error Handling (+3.9%), where recommended practice weights adjust contribution levels. Code Automation experiences an 11.8% increase, suggesting implementation of mandatory automation features beyond basic requirements.

4.6.4.2 Lower Scores under Best-Practice Policy

Conversely, the `genzouw.selinux-disable` repository exhibits quality score reduction under best-practice assessment. The total score decreases from 3.918 to 3.792, representing a 3.22% decline. Figure 4.12 presents the category-level variations.

The reduction manifests across multiple categories, with Code Maintainability showing the largest decrease from 0.166 to 0.1 (-39.8%). The Code Maintainability category comprises three attributes as defined in Table 4.2.

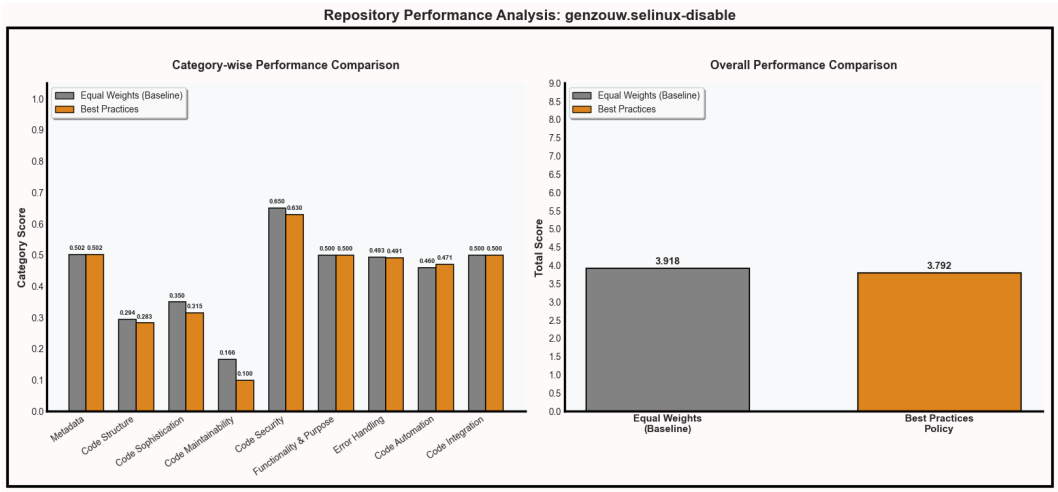


Figure 4.12: Lower Scores Under Best-Practice Configuration: `genzouw.selinux-disable` Repository

README word count and Comments are classified as mandatory ($w_{c,j} = 1.0$), while User interactions is classified as recommended ($w_{c,j} = 0.5$). The repository contains zero README files, zero words of documentation, zero inline comments, and zero user interaction attributes. Under equal weights, all three attributes contribute equally to the category score, producing a score of 0.166. Under best-practice weights, the mandatory classification of README word count and Comments amplifies the effect of their zero values, while the 0.5 weight on User interactions reduces its already zero contribution further relative to the mandatory attributes. This combination produces the -39.8% decrease. Code Structure decreases by 3.7%, as the repository contains only 25 lines of YAML across 4 files with no variables and no named tasks. Code Sophistication decreases by 10%, as the repository contains no loops, no conditionals, no regular expressions, and no lookups.

The repository name itself, `selinux-disable`, indicates functionality that contradicts security best practices. The repository receives a lower Code Security score (0.63 from 0.65) because disabling SELinux represents an anti-pattern in security-conscious infrastructure management. Error Handling and Code Automation show minor variations (-0.4% and +2.4%), and Metadata and Code Integration remain unchanged at 0.502 and 0.5.

4.6.4.3 Observed Score Variation Patterns

The comparative analysis reveals three patterns in policy-based assessment: repositories implementing mandatory practices receive higher scores proportional to their adherence level, those lacking mandatory attributes or implementing anti-patterns receive reduced scores, and categories with recommended practices show bidirectional movements based on implementation completeness. The `elkstack` repository's higher score reflects alignment with mandatory documentation and security practices, while `selinux-disable`'s reduction demonstrates penalisation for minimal documentation and anti-security purpose. Code Sophistication variations in both repositories (-4.5% and -10%) illustrate how 0.5 weights on recommended features adjust their contribution relative to equal weighting. Score variations ranging from -39.8% to +20% at category level confirm the policy mechanism's sensitivity in differentiating quality implementations from minimal or anti-pattern code, validating its effectiveness for practical IaC quality assessment.

4.7 Category-Level Policy Evaluation and Results

The preceding sections evaluated how attribute-level policies affect quality scores across the repository dataset. This section extends the evaluation to the category level, where development philosophies defined in Section 4.4.1 determine the relative importance of entire quality dimensions. The analysis examines how repositories align with the eight development philosophies, revealing differential patterns between organisational and individual development practices across both single-point comparisons and longitudinal evolution analyses.

For this evaluation, the best-practice attribute-level weight configuration is applied as the foundation for category-level assessment. Consistent with the

attribute-level evaluation, all code-based attributes are normalised per 100 lines of code to enable fair comparison across repositories of varying sizes. This normalisation prevents larger repositories from receiving higher scores solely because they contain more absolute instances of a given attribute. The same exceptions described in the attribute-level evaluation apply here, where metadata attributes and size-invariant metrics are exempt from this normalisation.

4.7.1 Repository Performance Across Development Philosophies

The application of category-level policies to the 26,986 Ansible Galaxy repositories reveals differential alignment patterns between organisational and individual developer contributions. To illustrate these patterns, two repositories implementing the same security benchmark (RHEL 7 CIS) were selected for comparison. The official Red Hat repository (`RedHatOfficial.rhel7_cis`) represents organisational development practices, while `HarryHarcourt._RHEL7-CIS` represents an individual contributor's implementation of the same benchmark. Selecting two repositories that target identical functionality isolates the effect of development practices on philosophical alignment, as differences in scores cannot be attributed to differences in purpose or scope.

In this evaluation, repository scores are expressed as alignment percentages rather than raw quality scores. The alignment percentage is computed using the alignment score formula defined in Section 4.2.2, where each repository's category scores are weighted according to the selected development philosophy and the result is scaled to a 0–100 range. A higher alignment percentage indicates that a repository's quality profile corresponds more closely to the priorities encoded in a given philosophy.

Figure 4.13 presents a radar chart comparing the two repositories against the dataset average across the eight development philosophies. Each axis of the radar chart represents one development philosophy, and the distance from

the centre along each axis corresponds to the alignment percentage for that philosophy. The shaded polygon for each repository therefore shows its alignment profile across all eight philosophies, allowing visual comparison of where repositories converge and diverge.

The dataset average alignment percentage varies between 51.4% and 54.3% depending on the philosophy, as different philosophies weight different categories and therefore produce different average scores across the repository population. Both repositories exceed the dataset average across all philosophies, with alignment percentages ranging from 54.3% to 60.8%. The two repositories achieve similar alignment in speed-focused philosophies (Move Fast & Break Things, Minimum Viable Product, Lean Infrastructure), where both score between 57.1% and 58.3%. However, divergence emerges in specialised excellence and comprehensive quality philosophies, where the official Red Hat repository consistently scores higher than the individual user's implementation.

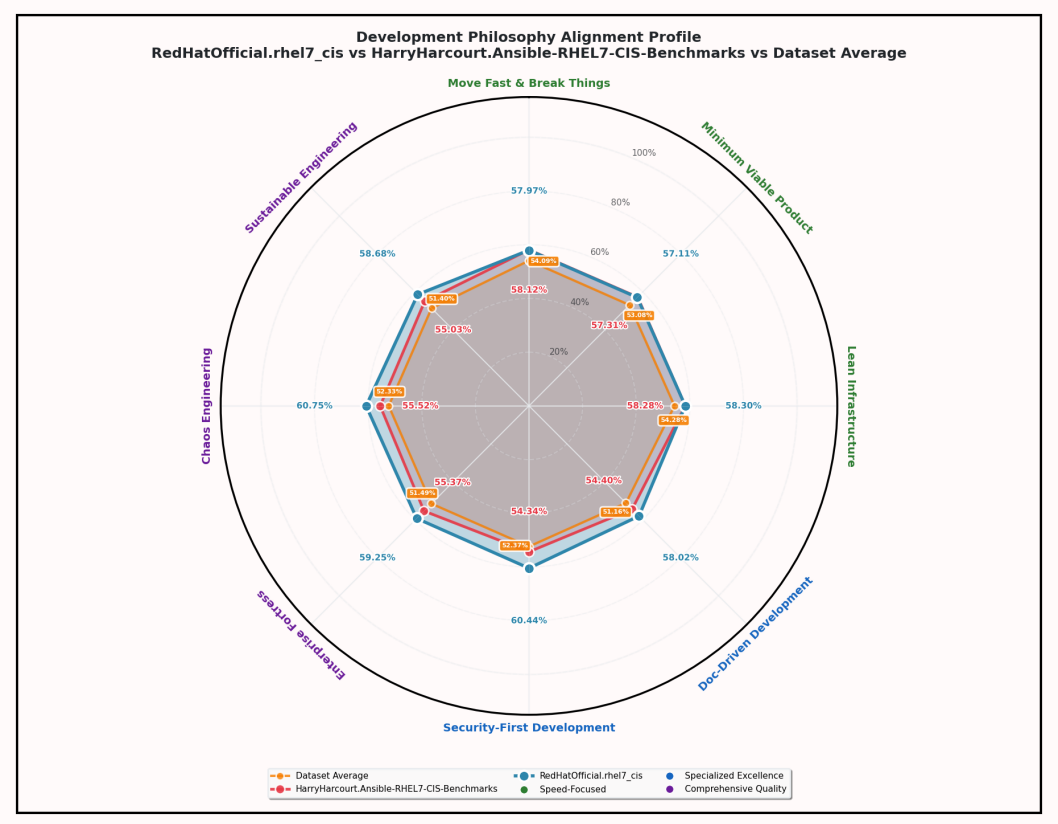


Figure 4.13: Category-Level Policy Alignment: RHEL 7 CIS Organisational and Individual Repository Comparison With Dataset Baseline

The performance gap between the official Red Hat repository and the individual user's implementation widens when we consider the philosophies that emphasise long-term sustainability and operational maturity. The official `RedHatOfficial.rhel7_cis` achieves 60.4% alignment with the Security-First Development philosophy compared to 54.3% for the `HarryHarcourt` implementation, representing a 6.1 percentage point difference. This gap persists across Enterprise Fortress (59.3% vs 55.4%), Chaos Engineering (60.8% vs 55.5%), and Sustainable Engineering (58.7% vs 55.0%) philosophies. The official repository demonstrates 4–6 percentage points higher alignment with philosophies that prioritise security, reliability, and maintainability. This pattern aligns with findings from the attribute-level analysis, suggesting that repositories maintained by organisations invest in quality dimensions required by enterprise philosophies, while individual contributors optimise for use cases that align with speed-focused development. Given that organisational repositories would be expected to score higher on enterprise-oriented philosophies, and individual repositories would be expected to score higher on speed-focused philosophies, this outcome serves to validate both the category-level policy mechanism and the framework itself.

4.7.2 Repository Quality Evolution at Category Level

While the previous subsection examined alignment patterns between organisational and individual repositories at a single temporal point, the examination of code quality evolution across development lifecycles provides insights into how repositories transform their philosophical alignment through sustained development. To conduct this examination, a mechanism is needed for observing how repositories have changed over time. Version control systems provide this mechanism through commit histories, where each commit represents a recorded state of the repository at a specific point in its development. The longitudinal analysis of these commit histories enables identification of code quality shifts as projects mature from initial implementations through active development

to stable production states. This subsection presents a comparative analysis of two nginx repositories across three development stages, examining how organisational and individual development practices manifest different philosophical evolution trajectories.

The analysis employs commit-level assessment where each commit undergoes evaluation against the eight development philosophies defined in Section 4.4.1. To examine how quality evolves across the lifecycle of each repository, commit histories are divided into three equal stages. The Initial stage contains the first 33% of commits, the Development stage contains the middle 34%, and the Present stage contains the final 33%. This division allows comparison of quality patterns across repositories that have different numbers of commits and different development timelines. The official `nginx.nginx` repository contains 805 commits from 62 contributors spanning 2018 to 2025, while the `geerlingguy.nginx` repository contains 245 commits from a single developer spanning 2014 to 2025. The divergent commit distributions reflect differences in development approach. Organisational repositories engage in continuous refinement through frequent incremental updates distributed across the development timeline, whereas individual repositories concentrate development effort during initial phases followed by declining commit frequency as projects transition toward maintenance mode.

4.7.2.1 Official nginx Repository Quality Evolution

This subsubsection examines how the official `nginx.nginx` repository's alignment with the eight development philosophies changed across its 805 commits. The purpose of this examination is to observe whether the repository's quality profile shifted between philosophy categories as the project matured through its three development stages.

In the Initial stage, the repository exhibits highest alignment with speed-focused philosophies. Move Fast achieves 49.11% alignment and Lean Infrastructure achieves 44.00%, while comprehensive quality philosophies score be-

tween 32.17% and 35.12%. The difference between the highest and lowest philosophy alignment scores in this stage is 16.94 percentage points, indicating that the repository's quality profile in its early commits aligns unevenly across philosophies, with a clear orientation toward speed-focused development.

In the Development stage, the alignment scores shift toward comprehensive quality philosophies. Security-First Development emerges as the philosophy with the highest alignment (61.72%), representing a 28.09 percentage point increase from the Initial stage. Enterprise Fortress reaches 59.99%, and the overall mean alignment across all philosophies increases from 44.68% to 54.14%. These changes suggest that the repository's development activity during this stage introduced practices that correspond to security and enterprise quality requirements.

In the Present stage, Documentation-Driven Development records the highest alignment (69.10%), followed by Minimum Viable Product (68.71%) and Enterprise Fortress (68.23%). The overall mean philosophy alignment reaches 64.92%, representing a 20.24 percentage point increase from the Initial stage. All philosophies exceed 62% alignment in this stage, and the dispersion between the highest and lowest philosophy scores narrows compared to the Initial stage. This pattern suggests that the repository's quality profile has become more balanced across philosophical dimensions, rather than being concentrated in one category of philosophy. The high alignment scores recorded for both Documentation-Driven Development and Enterprise Fortress indicate that the repository's present state reflects practices associated with documentation quality, production-grade reliability, and feature development, though the framework measures alignment with these philosophies rather than the developers' stated intentions.

Figure 4.14 presents a radar chart comparing philosophy alignment scores across the three development stages for the official nginx repository. The expansion of the radar polygon from the Initial stage through the Development stage to the Present stage illustrates increasing alignment scores across

all philosophical dimensions. While Figure 4.14 presents a static comparison of the three stages, an animated visualisation that renders the alignment changes at each individual commit is available in the supplementary materials at GitHub.[20] This animation shows how the radar polygon expands and contracts at each commit, revealing intermediate fluctuations in alignment that are not visible in the three-stage summary presented in this thesis.

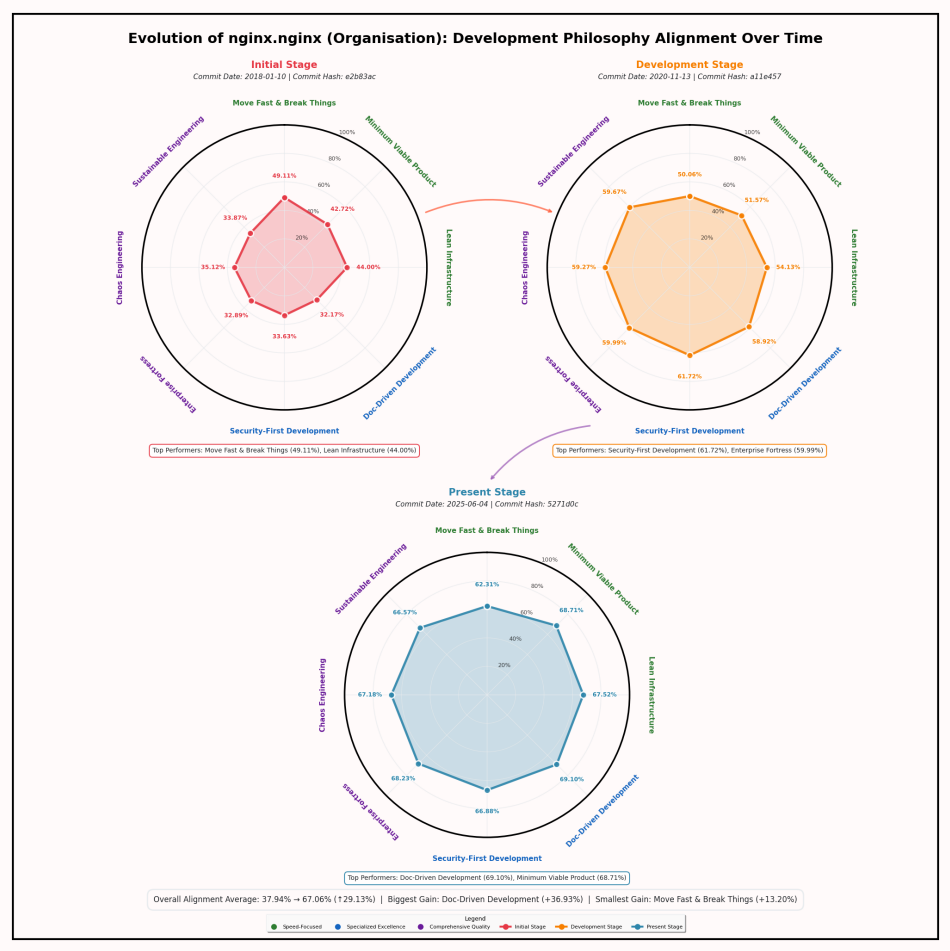


Figure 4.14: Organisational Repository Philosophy Evolution: Official Nginx Repository Across Three Development Stages (enlarged versions are provided in Appendix A, Figure A.1, Figure A.2, and Figure A.3)

In addition to examining alignment scores at each stage, we analyse how each philosophy's alignment deviates from the stage mean. This deviation analysis measures whether a repository's quality profile favours certain philoso-

[20] Pandu Ranga Reddy Konala, Vimal Kumar, David Bainbridge, and Junaid Haseeb, "Understanding Repository Philosophy Evolution: Official Nginx Repository (Github)," 2025, https://github.com/PanduKonala/ansible-quality-evolution-visualizations/blob/659e2ddbde1820b8eeef81485b41f5b5df9fe4b1/animations/nginx_official_evolution.mp4.

phies over others within each stage. By comparing these deviations across the three stages, we can observe whether the philosophies that score above or below the mean change as the repository matures. If the repository's development practices shift over time, we would expect the philosophies with positive deviations to change between stages. Figure 4.15 presents these deviations for the official nginx repository.

During the Initial stage, the mean alignment across all philosophies is 44.68%. Move Fast exhibits the highest positive deviation from this mean at +7.83 percentage points, reaching 52.52%. Lean Infrastructure follows with a deviation of +3.64 points from the mean, reaching 48.32%. All comprehensive quality philosophies fall below the stage mean. Documentation-Driven Development deviates by -3.60 points from the mean at 41.08%, Enterprise Fortress by -3.24 points at 41.44%, and Security-First Development by -3.19 points at 41.49%. These deviations indicate that the repository's early commits align more closely with speed-focused philosophies than with comprehensive quality philosophies.

During the Development stage, the mean alignment increases to 54.14% and the pattern of deviations reverses. Security-First Development achieves the highest positive deviation from the mean at +1.97 points, reaching 56.10%. Enterprise Fortress and Chaos Engineering each deviate by +0.95 points from the mean, reaching 55.09%. Speed-focused philosophies shift to negative deviations, with Move Fast at -2.23 points from the mean at 51.91% and Minimum Viable Product at -2.14 points at 51.99%. This reversal suggests that the development activity during this stage introduced practices that correspond to security and enterprise requirements rather than speed-focused development.

During the Present stage, the mean alignment reaches 64.92%. Documentation-Driven Development records the highest positive deviation from the mean at +2.65 points, reaching 67.57%. Enterprise Fortress follows at +1.68 points, reaching 66.60%. Move Fast records the largest negative deviation at -5.24 points from the mean, reaching 59.68%.

Across the three stages, the philosophies with positive deviations shift from speed-focused (Move Fast, Lean Infrastructure) in the Initial stage to comprehensive quality philosophies (Security-First, Enterprise Fortress, Documentation-Driven) in the Development and Present stages. This shift indicates that the repository's quality profile changed over time, moving from alignment with speed-focused development toward alignment with philosophies that emphasise security, documentation, and maintainability.

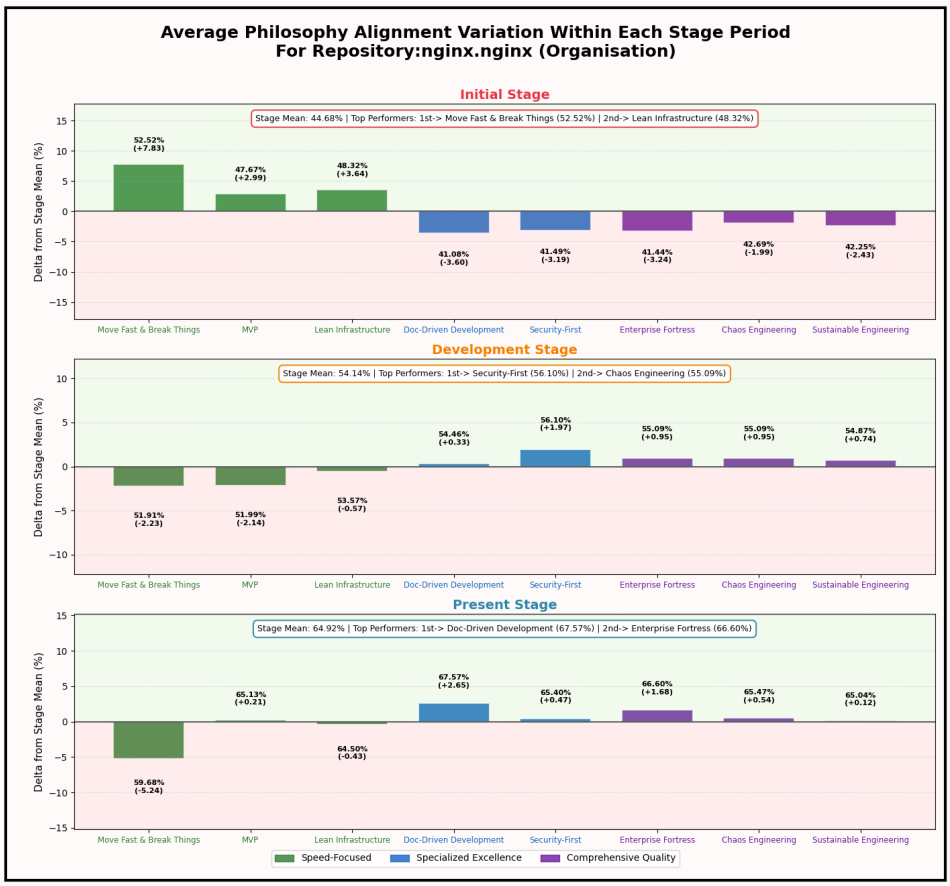


Figure 4.15: Philosophy Alignment Deviations Across Development Stages: Official Nginx Repository

4.7.2.2 Individual Developer Repository Quality Evolution (geerlingguy.nginx)

This subsubsection examines how the `geerlingguy.nginx` repository's alignment with the eight development philosophies changed across its 245 commits. As noted in the previous subsubsection, the official nginx repository shifted

from speed-focused to comprehensive quality philosophies over its development lifecycle. The purpose of this examination is to observe whether an individual developer's repository follows a similar or different trajectory, and where any divergence occurs.

In the Initial stage, the `geerlingguy.nginx` repository exhibits a philosophy alignment pattern that is comparable to the official repository's Initial stage. Move Fast achieves 50.33% alignment and Lean Infrastructure achieves 47.30%, while comprehensive quality philosophies remain below 41.80%. This similarity is worth noting because the two repositories differ in authorship type, contributor count, and development timeline, yet both begin with a quality profile oriented toward speed-focused philosophies. This observation suggests that speed-focused development may be a common starting pattern for Ansible repositories regardless of whether they are maintained by organisations or individuals.

In the Development stage, the mean alignment across all philosophies increases to 67.29%, a 16.15 percentage point increase from the Initial stage. Documentation-Driven Development emerges as the philosophy with the highest alignment at 69.26%, and Enterprise Fortress reaches 66.75%. These changes indicate that the repository's development activity during this stage introduced practices that correspond to documentation and enterprise quality requirements, following a similar trajectory to the official repository.

In the Present stage, the trajectory diverges from the official repository. Enterprise Fortress becomes the philosophy with the highest alignment at 72.13%, followed by Chaos Engineering at 71.34% and Sustainable Engineering at 71.27%. Speed-focused philosophies show minimal increases in the Present stage compared to earlier stages. Move Fast accumulates the smallest cumulative gain across all three stages at +19.14 percentage points, while Enterprise Fortress accumulates the largest at +33.01 points. This pattern suggests that the repository's recent commits focused on maintaining and consolidating existing practices rather than introducing new functionality, which is consistent

with a single-developer repository entering a maintenance phase.

Figure 4.16 presents a radar chart comparing philosophy alignment scores across the three development stages for the `geerlingguy.nginx` repository. The radar polygon expands between the Initial and Development stages, reflecting the increase in alignment scores described above. Between the Development and Present stages, the polygon expands by a similar amount across all philosophy axes, indicating that alignment scores increased across all philosophies rather than in selected dimensions. While Figure 4.16 presents a static comparison of the three stages, an animated visualisation that renders the alignment changes at each individual commit is available in the supplementary materials at GitHub.[21] This animation shows how the radar polygon expands and contracts at each commit, revealing intermediate fluctuations in alignment that are not visible in the three-stage summary presented in this thesis.

The examination of stage-to-stage philosophy alignment changes for the `geerlingguy.nginx` repository is presented in Figure 4.17. During the Initial stage, the mean alignment across all philosophies is 51.14%. The deviations from this mean range from -1.05 to +2.10 percentage points, a narrow range that indicates no single philosophy dominates the repository's quality profile at this stage. Lean Infrastructure achieves the highest positive deviation from the mean at +2.10 percentage points, reaching 53.24%, followed by Move Fast at +0.74 points from the mean at 51.88% and Minimum Viable Product at +0.70 points at 51.84%, while comprehensive quality philosophies cluster slightly below the stage mean with deviations ranging from -1.05 to -0.25 points, suggesting a more balanced initial quality profile compared to the official repository's Initial stage. During the Development stage, the mean alignment increases to 67.29%. Documentation-Driven Development achieves the highest positive deviation from the mean at +1.96 points, reaching 69.26%,

[21] Pandu Ranga Reddy Konala, Vimal Kumar, David Bainbridge, and Junaid Haseeb, "Understanding Repository Philosophy Evolution: geerlingguy.nginx by Solo Developer Jeff Geerling (Github)," 2025, https://github.com/PanduKonala/ansible-quality-evolution-visualizations/blob/659e2ddbde1820b8eeef81485b41f5b5df9fe4b1/animations/geerlingguy_evolution.mp4.

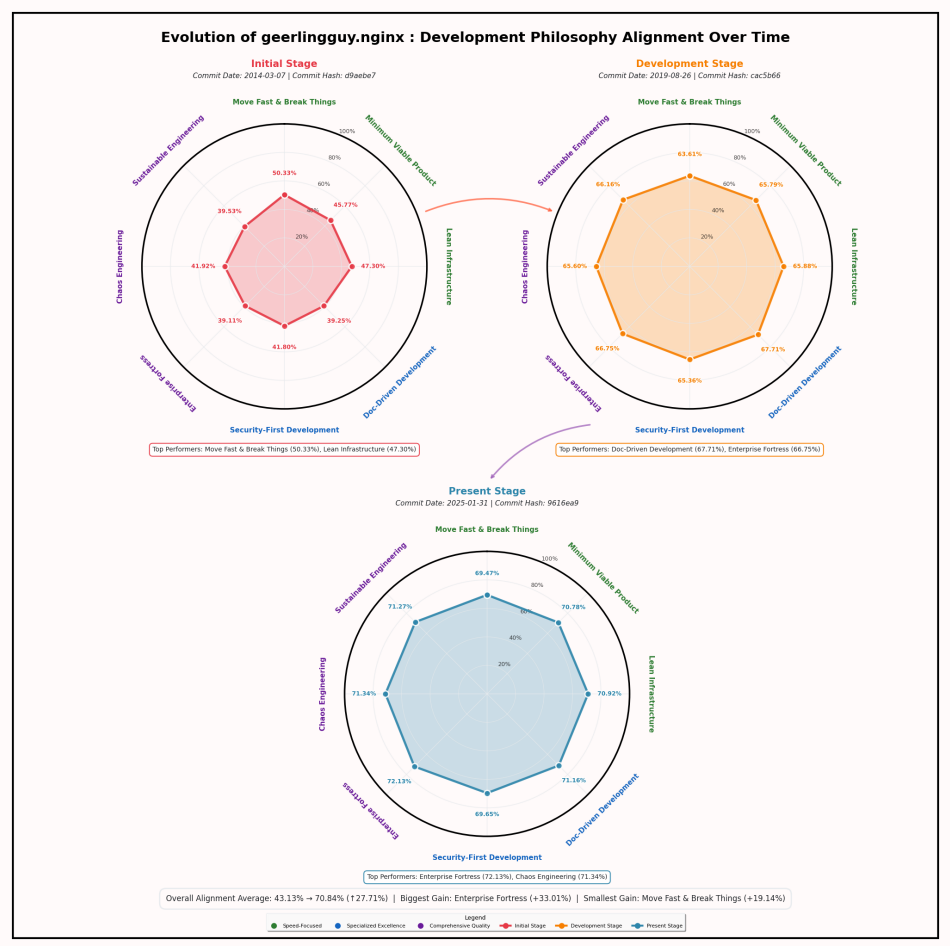


Figure 4.16: Individual Repository Philosophy Evolution: `geerlingguy.nginx` Repository Across Three Development Stages (enlarged versions are provided in Appendix A, Figure A.4, Figure A.5, and Figure A.6)

and Enterprise Fortress at +1.28 points from the mean at 68.57%, while Move Fast shifts to the largest negative deviation at -2.48 points from the mean at 64.82%, indicating that development activity during this stage introduced practices corresponding to documentation and enterprise requirements. During the Present stage, the mean alignment reaches 70.72%. Enterprise Fortress achieves the highest positive deviation from the mean at +1.60 points, reaching 72.32%, followed by Documentation-Driven Development at +1.31 points at 72.03%, Sustainable Engineering at +0.67 points at 71.39%, and Chaos Engineering at +0.38 points at 71.09%, while Move Fast maintains the largest negative deviation at -2.64 points from the mean at 68.07%, suggesting that the repository's recent commits focused on maintaining existing practices rather

than introducing new functionality.

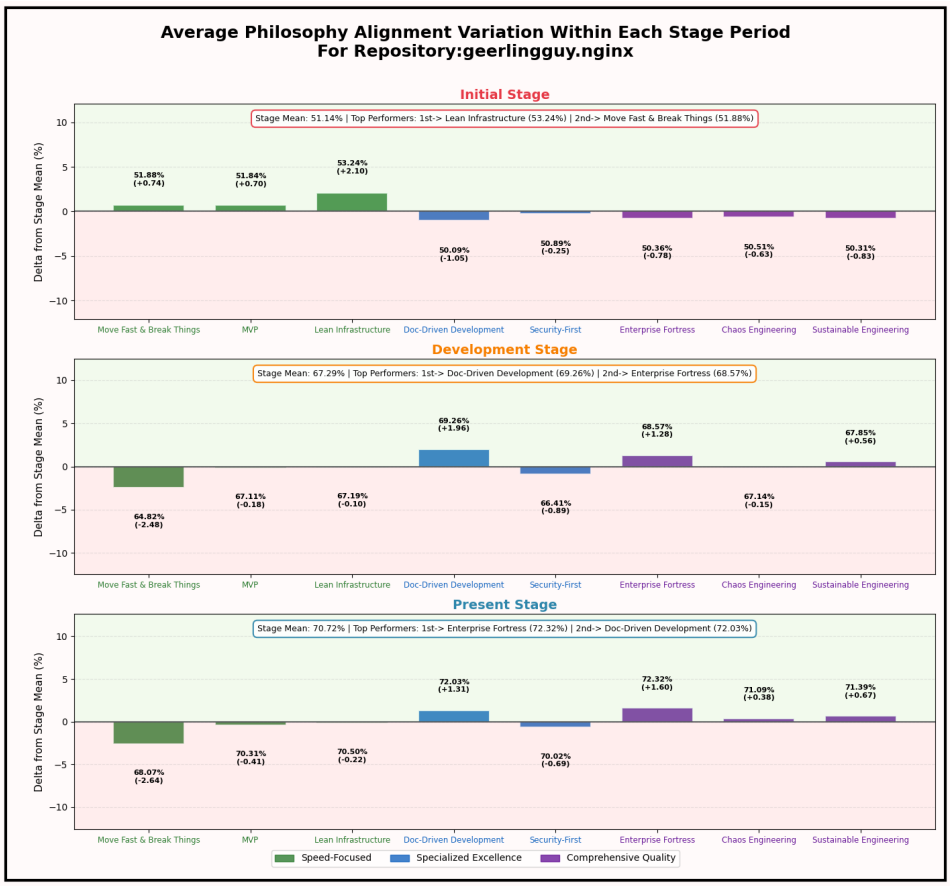


Figure 4.17: Philosophy Alignment Deviations Across Development Stages: Individual Nginx Repository

The longitudinal analysis of nginx repository evolution across development philosophies establishes that repository quality trajectories reflect development practices rather than random accumulation. The examination of category-level policy applications through both single-point comparisons (RHEL CIS repositories) and temporal evolution (nginx repositories) validates the framework's capacity to quantify philosophical alignment and track quality changes over time. The deviation analysis presented in Figures 4.15 and 4.17 compared how each philosophy's alignment score deviated from the stage mean across the three development stages for both repositories. This comparison reveals that organisational and individual repositories follow different trajectories. In the official repository, the Initial stage exhibited a wide deviation range with speed-focused philosophies scoring above the mean, which then reversed in

later stages as comprehensive quality philosophies moved above the mean. In the `geerlingguy.nginx` repository, the Initial stage exhibited a narrow deviation range, indicating a more balanced starting profile, followed by a similar shift toward comprehensive quality philosophies in later stages. These findings demonstrate the framework's utility for both cross-repository assessment and longitudinal quality monitoring within organisational contexts. The subsequent section synthesises the empirical findings to discuss broader implications for IaC quality assessment practices and identifies limitations requiring consideration in framework deployment.

4.8 Discussion

This section discusses the findings from the empirical evaluation of the IaC quality framework's policy mechanism, applied to 26,986 Ansible Galaxy repositories. The discussion examines how the two types of policy configurations, attribute-level and category-level, affect quality assessment outcomes and what patterns emerge from their application.

The attribute-level policy comparison between equal-weight and best-practice configurations shows that the aggregate difference in total scores between the two configurations is small at 1.5%. However, at the level of individual categories, the differences are larger. Functionality & Purpose scores differ by 10.7% between the two configurations, and Code Maintainability scores differ by 3.8%. These category-level differences indicate that the policy mechanism selectively enforces community standards in categories where mandatory practices and anti-patterns are defined, while producing similar scores in categories where repositories naturally follow consistent practices.

The temporal evaluation from 2014 to 2023 demonstrated consistent trends in quality score increases across the platform. When examining repositories individually, the results indicated that organisational repositories follow distinct development paths. Organisational repositories show score increases dis-

tributed across multiple quality categories over successive commits, while individual repositories show concentrated score increases in early commits followed by stable scores in later commits as the repository enters a maintenance phase.

Category-level policy was used to quantify the philosophical alignment of repositories. Organisational repositories score higher on enterprise-oriented philosophies, while individual repositories show balanced alignment across all philosophical dimensions. Together with the attribute-level findings, these results demonstrate that the framework can assess IaC quality across different organisational contexts. The attribute-level policy mechanism differentiates repositories based on adherence to community standards, while the category-level policy mechanism differentiates repositories based on alignment with organisational development philosophies. The combination of these two mechanisms enables organisations to configure the framework to reflect their priorities at both levels.

4.8.1 Effects of Policy Choice on Quality Evaluation

This subsection discusses how the choice of policy configuration affects the quality scores produced by the framework, drawing on the attribute-level and category-level evaluations presented in the preceding sections.

At the attribute level, the comparison between equal-weight and best-practice configurations reveals that the policy mechanism produces larger score differences in categories where mandatory practices and anti-patterns are defined. Functionality & Purpose scores are 10.7% higher under best-practice weights due to the mandatory weighting of documentation and syntax validation attributes. Error Handling scores are 3.2% higher due to the penalisation of the `ignore_errors` anti-pattern. Conversely, categories containing mostly recommended practices receive lower scores under best-practice weights. Code Maintainability scores decrease by 3.8% and Code Sophistication scores decrease by 2.9%, as the 0.5 weight assigned to recommended attributes reduces their contribution relative to the equal-weight configuration. This pattern in-

dicates that best-practice weights amplify established community standards in categories with mandatory and discouraged practices, while producing smaller differences in categories where practices are context-dependent.

The single-repository analyses presented in Section 4.6.4 further illustrate this pattern. The `mrlesmithjr.elkstack` repository receives scores that are 4.7% higher under best-practice weights because its implementation aligns with mandatory structural and security attributes. In contrast, the `genzouw.selinux-disable` repository receives scores that are 3.22% lower under best-practice weights because it contains minimal documentation and its purpose of disabling SELinux corresponds to a discouraged security practice.

At the category level, alignment scores indicate that organisational repositories correspond more closely to enterprise-oriented philosophies than individual repositories. Red Hat's official repository (`RedHatOfficial.rhel7_cis`) achieves the highest alignment across Security-First Development, Enterprise Fortress, Chaos Engineering, and Sustainable Engineering philosophies. Individual repositories achieve comparable alignment with speed-focused philosophies but lower alignment with enterprise-oriented philosophies. This difference suggests that organisational development processes produce quality profiles that correspond to enterprise philosophy requirements across multiple categories, whereas individual repositories concentrate on categories associated with rapid deployment. The framework provides organisations with quantitative measures of whether their infrastructure code repositories align with their intended development philosophies, enabling informed decisions about repository selection and quality investment.

4.8.2 Use Cases of the IaC Quality Framework

The framework's two-level policy structure supports two distinct use cases that correspond to the two levels of policy configuration.

At the category level, organisations can use philosophy alignment scores to evaluate whether their repositories conform to the organisation's develop-

ment philosophy. For example, an organisation that adopts a Security-First philosophy can assess all repositories in its portfolio against that philosophy's category weights and identify repositories where alignment scores fall below the organisation's threshold. This enables organisations to detect repositories that have deviated from their preferred practices before such deviations affect operations.

At the attribute level, the framework provides developers with feedback on specific quality attributes within their code. The evaluation of 26,986 repositories found that 89.7% lacked proper structural organisation and only 8.26% implemented comprehensive error handling. These findings identify specific areas where developers can direct their efforts, with the framework providing quantitative measures for tracking progress.

Organisations may begin with an equal-weight configuration to assess the current state of their codebase, then subsequently apply best-practice weights based on industry standards or custom configurations that reflect organisational priorities as they mature. Platform maintainers could utilise the framework to display quality scores alongside other repository metadata, creating transparency that encourages adoption of best practices and enables informed decision-making when selecting repositories aligned with the organisation's development philosophy. The small overall variability between policy configurations for well-maintained repositories demonstrates that well-written infrastructure code generally adheres to industry standards, confirming that the framework's quality metrics capture genuine quality characteristics rather than imposing arbitrary requirements.

4.8.3 Generalisability to Other Platforms

The multi-platform claim of Chapter 3 attaches to different parts of the framework in different ways. The nine categories and the scoring algorithm are platform-neutral by construction. The categories derive from ISO/IEC 25010 together with attribute catalogues spanning two platforms, and the mapping

of Rahman et al.'s Puppet attributes in Chapter 3 shows that the categories accommodate a platform this evaluation never touched. What is platform-specific by design are two adapter layers: the attribute extractor, which parses the syntax of the target platform, and the weight configuration, which encodes the standards of the target community. The attributes themselves remain obtainable to different degrees. Code-level attributes such as loops, conditionals, and error handling constructs have equivalents in Terraform configurations and Kubernetes manifests, and require a new extractor rather than new categories. Metadata attributes depend on the platform hub. The Terraform Registry, Chef Supermarket, and Artifact Hub expose fields comparable to those Ansible Galaxy enforces, while a platform without a central hub would operate with a reduced attribute set in the Metadata category.

The costlier half of a port is the weight configuration, because deriving best-practice weights requires documented community standards, and ecosystems with fragmented conventions, such as Kubernetes and Docker, make that codification the harder part of the transfer. The applicability of the framework therefore transfers by construction, while the empirical findings of this chapter do not. The quality distributions and temporal trends reported here are facts about Ansible Galaxy, and other ecosystems with different architectures and community practices may behave differently. Establishing the findings on another platform requires replication, which remains future work. The evaluation concentrated on a single platform studied in depth because the subsequent chapters build on the same validated substrate, and the Puppet mapping provides low-cost evidence of transferability.

4.8.4 Limitations and Future Work

This subsection identifies the limitations of the policy-based evaluation presented in this chapter and outlines directions for future research.

The weight configurations used in this evaluation were derived through interpretation of platform documentation and community standards. This

derivation process involves subjectivity and does not always account for context-specific nuances. The Code Sophistication trend illustrates this limitation, as the observed decline in scores reflects changes in platform evolution rather than a decline in code quality.

The weight configurations derived in this chapter are also specific to Ansible's documentation and community standards. This platform specificity limits the current weight configurations rather than the framework, and Section 4.8.3 discusses the transfer path to other platforms .

Another limitation of this work is that comparisons of scores across different policy configurations do not provide direct evidence of quality differences , for the reasons given in Section 4.6.2 . Organisations must therefore maintain policy stability to enable longitudinal assessments and document the rationale behind their weight allocations to ensure transparency.

The manual effort required to derive weight configurations from platform documentation presents an opportunity for future research. Methods for automated weight derivation from platform documentation could reduce this effort, and investigation into the transferability of policies across IaC platforms could extend the framework's applicability beyond Ansible.

4.9 Chapter Summary

This chapter validated the policy-based IaC quality assessment framework through empirical analysis of 26,986 Ansible Galaxy repositories. We compared equal-weight and best-practice attribute-level configurations and found that the aggregate difference in total scores between the two configurations was small at 1.5%. However, at the category level, we observed differences of up to 10.7%, demonstrating that the policy mechanism selectively enforces community standards where mandatory practices and anti-patterns are defined.

Our analysis of temporal trends spanning 2014 to 2023 revealed consistent patterns of increasing quality scores across the platform. We also performed

longitudinal examination of individual repositories through their commit histories. This examination demonstrated that organisational repositories show score increases distributed across multiple quality categories over successive commits, while individual repositories show concentrated score increases in early commits followed by stable scores in later stages.

At the category level, the application of development philosophy weights quantified the philosophical alignment of repositories. Organisational repositories achieved higher alignment with enterprise-focused philosophies compared to individual implementations. Through these evaluations, we demonstrated that the framework can differentiate repositories through policy-driven assessment while maintaining consistency across configurations, providing evidence for its utility in alignment monitoring and quality assessment within diverse organisational contexts.

Chapter 5

Quality Based Attack Detection: Identifying Supply-Chain Threats[1]

Supply chain vulnerabilities in IaC repositories represent a challenging class of security concerns. A commonly used Ansible role, for example, may depend on a third-party module whose developer has unintentionally introduced a vulnerability through outdated configurations. In a more adversarial scenario, a malicious developer may deliberately obfuscate a vulnerability within a widely adopted dependency. Detecting such issues requires analysis of dependency chains, maintenance patterns, and author activity rather than code pattern matching. These elements can be collectively viewed as repository metadata, information that describes the context surrounding the code rather than the code itself.

As observed in Chapter 2, current IaC tools exhibit metadata blindness, performing their analysis on scripts individually rather than at repository level, and as a result cannot track cross-repository relationships. These limitations render conventional tools ineffective for supply chain threat detection, which depends on examining inter-repository dependencies and temporal patterns.

[1]This chapter largely builds upon P. R. R. Konala et al.'s research [67], incorporating revisions for thesis presentation.

The previous chapters established a framework for assessing the quality of IaC repositories through nine categories (Chapter 3) and using policies to enforce organisation priorities via weights (Chapter 4). By including the capability to analyse metadata as part of its systematic method for calculating quality scores, this framework addresses a limitation of previous studies that assessed IaC quality based solely on code-level attributes. The omission of metadata from quality assessment prevents identification of risks that originate outside the code itself, such as dependency vulnerabilities, abandoned maintenance, and compromised authorship. The metadata analysis capability of the framework therefore provides an opportunity to investigate whether the quality assessment approach can extend beyond code improvement to identify security threats overlooked by traditional analysis tools.

This chapter investigates whether the quality assessment framework can be used to help identify supply chain threats within IaC ecosystems. The investigation begins by examining supply chain attack incidents and reviewing existing research to establish the threat landscape. With this context in place, the analysis then evaluates current IaC smell detection tools to understand their capabilities and limitations regarding metadata processing. Building on these findings, the chapter develops a methodology that applies the quality framework with security-focused policy configurations to dependency chain analysis. Finally, we apply our methodology to real-world IaC repositories to examine what vulnerability patterns emerge and what implications they hold for IaC security practices.

5.1 Related Work

This section establishes the foundation for understanding supply chain attacks in both traditional software and IaC contexts. It examines historical incidents, reviews existing research approaches, and identifies gaps in current methodologies.

5.1.1 Notable Supply Chain Attack Incidents

Supply chain attacks demonstrate vulnerabilities across industries through various attack vectors. The Colonial Pipeline ransomware attack utilised a compromised password to disrupt fuel distribution in the United States, affecting petrol and jet fuel supply along the East Coast [68]. JBS, a meat producer, experienced a supply chain ransomware attack attributed to the REvil group, interrupting operations across Australia, Canada, and the United States [69].

Software-specific supply chain attacks reveal patterns relevant to IaC security. The 2018 attack on the Node.js *eslint-scope* package exemplifies dependency poisoning, where attackers exploited a stolen *npm* token to release a malicious version that exfiltrated *npm* tokens from dependent machines.[2] The 2022 attack on a WordPress plug-in and theme developer compromised 40 themes and 53 plug-ins, affecting over 360,000 websites [70].

The PyPI supply chain attack in 2024 targeted the Python Package Index, a repository for Python libraries.[3] Attackers uploaded packages that appeared legitimate while containing malware designed to exfiltrate information from compromised systems. This incident exposed weaknesses in open-source supply chains and emphasised the need for dependency verification prior to integration in software projects.

Recent incidents demonstrate that these attack patterns have reached the toolchains surrounding infrastructure code itself. In March 2025, attackers compromised *tj-actions/changed-files*, a CI/CD automation component used by more than 23,000 repositories, by retroactively re-pointing its version tags at a malicious commit. Because version tags were trusted as immutable references, downstream pipelines automatically executed the malicious code, which

[2]Henry Zhu, "Postmortem for Malicious Packages Published on July 12th, 2018," 2018, https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes/.

[3]JFrog Security Research, "Revival Hijack: PyPI Hijack Technique Exploited, 22K+ Packages at Risk," 2024, https://jfrog.com/blog/revival-hijack-pypi-hijack-technique-exploited-22k-packages-at-risk/.

exfiltrated CI/CD secrets into publicly readable build logs (CVE-2025-30066).[4] In March 2026, the TeamPCP campaign compromised the distribution channels of two widely deployed security scanners, pushing backdoored releases of the Trivy vulnerability scanner to Docker Hub and malicious commits to all 35 version tags of the GitHub action of KICS, an IaC analysis tool included in the survey in Chapter 2. The injected payloads harvested cloud credentials, CI/CD secrets, and SSH keys from the pipelines that invoked the scanners, while the compromised scanners continued to return normal results and so masked the theft.[5] The campaign inverted the trust relationship at the centre of the software supply chain, because the tools organisations deploy to detect vulnerable components became the vector through which their credentials were stolen.

For Ansible specifically, no in-the-wild attack on the Galaxy ecosystem has been publicly documented to date, yet the disclosure of CVE-2026-11332 in ansible-core demonstrated that the dependency mechanism itself carries the exposure. The flaw allows a malicious role author to inject git configuration flags through the `src` field of a role's `meta/requirements.yml` file, so that merely installing the role through `ansible-galaxy` executes attacker-controlled code on the installing machine.[6] The dependency declaration, the same metadata that the methodology presented in this chapter parses to construct dependency chains, is therefore itself an attack vector. The absence of documented Galaxy attacks reflects the absence of detection capability rather than the absence of exposure, because without tooling that examines dependency metadata such an attack would leave no recognisable trace.

[4]National Vulnerability Database, "CVE-2025-30066," March 2025, https://nvd.nist.gov/vuln/detail/CVE-2025-30066; GitHub Advisory Database, "GHSA-mrrh-fwg8-r2c3," March 2025, https://github.com/advisories/GHSA-mrrh-fwg8-r2c3.

[5]Palo Alto Networks Unit 42, "Weaponizing the Protectors: TeamPCP's Multi-Stage Supply Chain Attack on Security Infrastructure," March 2026, https://unit42.paloaltonetworks.com/teampcp-supply-chain-attacks/; Docker, "Trivy Supply Chain Compromise: What Docker Hub Users Should Know," March 2026, https://www.docker.com/blog/trivy-supply-chain-compromise-what-docker-hub-users-should-know/.

[6]National Vulnerability Database, "CVE-2026-11332," June 2026, https://nvd.nist.gov/vuln/detail/CVE-2026-11332.

These incidents establish recurring patterns in supply chain attacks: exploitation of trust relationships, targeting of widely used components, and abuse of automated dependency management . IaC systems exhibit every one of these characteristics through dependency declarations, package management, and trust-based distribution, and the incidents above show that the surrounding toolchain is already being targeted . The metadata and dependency information available in IaC repositories provides both the attack vectors and the detection opportunities that current security tools do not adequately address.

5.1.2 Software Supply Chain Security

Although the incidents above show the attack surface extending into infrastructure toolchains , research on supply chain security has predominantly focused on traditional software systems. Hossain Faruk et al. [71] investigated software supply chain security by examining vulnerabilities that adversaries exploit at multiple points in the software development lifecycle. Their study demonstrates that reliance on open-source or third-party code necessitates measures to protect the software supply chain from malicious activities. The authors identified issues in software supply chain security and reviewed existing methods and frameworks for securing the software supply chain, providing insights into prevention, detection, assessment, and remediation of security issues.

Duan et al. [72] examined package managers for interpreted languages, analysing PyPI (Python), *npm* (Node.js), and RubyGems (Ruby) to detect malicious packages. Their approach employed regular expressions to build heuristic rules derived from known supply chain attacks and malware studies, integrating static and dynamic code analysis. The study incorporated metadata-based heuristics including package names, authorship, popular packages with different authors, and version information. Dependency analysis identified vulnerabilities including single points of failure and risks associated with unmaintained packages.

Ohm et al. [73] reviewed 20 publications focused on identifying and detecting software supply chain attacks. The studies were categorised into six groups: rules and heuristics, typo-squatting, differential analysis, machine learning methods, anomaly detection, and clustering. Six of these studies integrated metadata analysis with static and dynamic code analysis, demonstrating the role of metadata in detecting supply chain attacks. The authors concluded that no single approach provides complete coverage; employing multiple strategies helps identify malicious behaviours across environments. They emphasised the need for future research adopting multimodal approaches for detecting security-breaching packages.

The transition from traditional software security to IaC security requires understanding how established principles apply to infrastructure management contexts. While existing studies focus on software code and security incidents, the use of IaC for building, orchestrating, and maintaining software systems necessitates investigation into how supply chain attacks manifest in IaC and how they can be detected .

5.2 IaC Tool Capabilities for Supply Chain Detection

Section 2.3.2 surveyed static analysis tools for IaC smell detection, examining their general capabilities and coverage. This section revisits those tools with a specific focus on their ability to process metadata and detect supply chain vulnerabilities. The examination analyses the levels of granularity at which these tools operate and evaluates whether any of them incorporate metadata retrieved from IaC codebase APIs into their analysis.

5.2.1 Analysis Granularity in IaC Tools

Static analysis tools for IaC scripts differ in the scope of input they process, which determines the types of vulnerabilities they can detect. Following the

approach of Ohm et al. [73] who surveyed traditional software tools, this analysis examines how these IaC tools process data to identify security smells. The examination identifies two levels of analysis in static vulnerability assessment: single file analysis and repository-level analysis. Figure 5.1 illustrates these analysis levels.

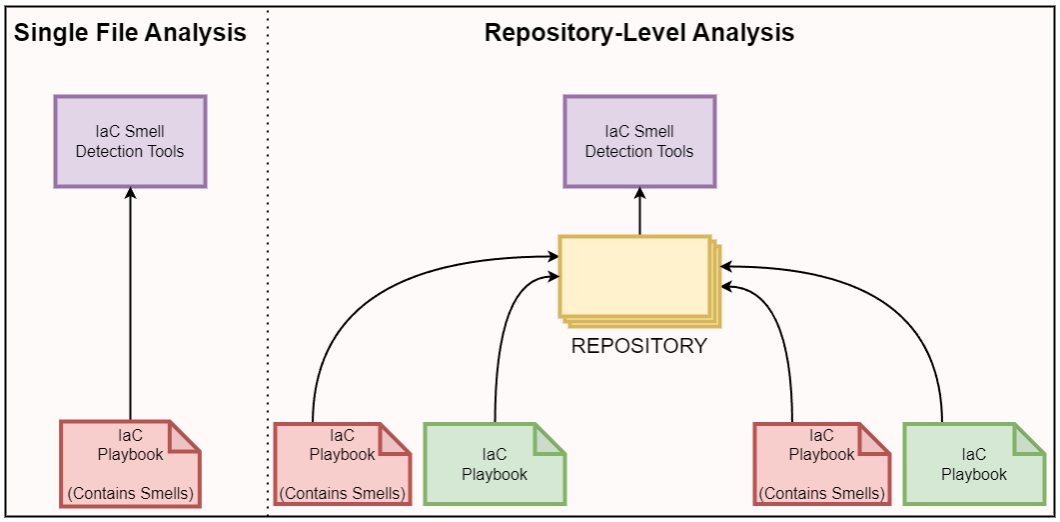


Figure 5.1: Analysis Granularity Levels in IaC Static Analysis Tools

5.2.1.1 File-Level Analysis

File analysis evaluates individual IaC files in isolation, identifying localised misconfigurations, security smells, and syntactical errors. This level examines hardcoded secrets, insecure configurations, and non-compliant syntax within individual files. Tools such as `ansible-lint` [7] and `yamllint` [8] verify Ansible script conformance to standards, while policy-as-code frameworks including Checkov [26] and KICS [31] ensure files meet predefined security and compliance benchmarks.

This approach operates without considering interdependencies between files or operational context. It establishes a baseline for file-level security, addressing vulnerabilities before integration into larger systems. The limitation of

[7] Will Thames and contributors, "ansible-lint: Ansible Playbook Linter," 2023, https://ansible-lint.readthedocs.io.

[8] Adrien Cunin, "yamllint Documentation: A Linter for YAML Files," 2023, https://yamllint.readthedocs.io.

this approach lies in its inability to detect issues arising from file interactions or dependency relationships.

5.2.1.2 Repository-Level Analysis

Repository-level analysis expands the scope to assess entire repositories as cohesive units. This level detects cross-file issues not identifiable through single file analysis, including variable reuse conflicts, configuration inconsistencies, and role or module misconfigurations. Policy-as-code tools such as Open Policy Agent (OPA) [9] and Checkov [26] enforce compliance and maintain consistency with organisational standards at this level.

By analysing interactions between IaC components, repository-level analysis provides insights into security issues emerging from multiple file operations. However, this level still operates within repository boundaries without examining external dependencies or metadata relationships. Whether existing IaC analysis tools overcome these limitations through metadata-aware analysis remains an open question.

This granularity distinction also determines the level at which the analysis in this chapter operates. Supply chain threats are properties of relationships between repositories. A dependency is declared once for the entire repository, in the role metadata (`meta/main.yml`), and binds the repository as a unit to its upstream components, so no individual file exhibits the dependency relationship that a supply chain attack exploits. File-level granularity therefore cannot express the threat model under study, and the findings of this chapter are reported at the repository level. File-level detection is not omitted but subsumed, because the security score of a repository aggregates the smell detections made in its individual files before Phase 3 of the methodology applies the threshold to the aggregated score.

[9] Open Policy Agent, "Open Policy Agent (OPA)," 2025, https://www.openpolicyagent.org/.

5.2.2 IaC Tool Capabilities and Limitations

To answer this question, the static IaC smell detection tools surveyed in Section 2.3.2, were systematically examined for their analytical mechanisms. The investigation addressed two specific questions: what levels of analysis are implemented by these tools, and whether the tools analyse metadata parameters retrieved from IaC codebase APIs.

Table 5.1: Analysis Granularity and Metadata Processing Capabilities of IaC Static Analysis Tools

Tool Name	Year	IaC Software Category	Availability	Levels of Analysis	Metadata Analysis
ACID[11]	2020	Configuration Management	Opensource	Repository Level	No
BARREL[25]	2018	Infrastructure Provisioning	Opensource	Single File	No
⋆ Checkov[26]	2021	Infrastructure Provisioning	Hybrid	Single File, Repository Level	No
⋆ CloudSploit[27]	2020	Infrastructure Provisioning	Proprietary	Repository Level	Unknown
CookStyle[28]	2016	Configuration Management	Opensource	Single File, Repository Level	No
DeepIaC[9]	2020	Configuration Management	Opensource	Single File	No
Foodcritic[29]	2011	Configuration Management	Opensource	Repository Level	No
GLITCH[15]	2022	Configuration Management	Opensource	Single File, Repository Level	No
Häyhä[30]	2021	Infrastructure Provisioning	Opensource	Single File	No
⋆ KICS[31]	2020	All	Opensource	Single File	No
Puppeteer[22]	2016	Configuration Management	Opensource	Repository Level	No
RADON[32]	2021	Configuration Management	Opensource	Single File, Repository Level	No
Rehearsal[10]	2016	Configuration Management	Opensource	Repository Level	No
SecGuru[33]	2014	Infrastructure Provisioning	Proprietary	Unknown	No
SecureCode[34]	2020	Configuration Management	Proprietary	Repository Level	No
⋆ Semgrep[35]	2020	All	Opensource	Single File	No
SLAC[36]	2021	Configuration Management	Opensource	Repository Level	No
SLIC[8]	2019	Configuration Management	Opensource	Repository Level	No
⋆ SNYK IaC[37]	2023	Infrastructure Provisioning, Image Building	Proprietary	Single File, Repository Level	No
SODALITE[38]	2020	Infrastructure Provisioning	Opensource	Single File, Repository Level	No
Sommelier[39]	2017	Infrastructure Provisioning	Opensource	Single File	No
⋆ SonarQube[40]	2016	Infrastructure Provisioning, Image Building	Hybrid	Repository Level	No
TAMA[41]	2022	Configuration Management	Opensource	Repository Level	No
⋆ tfsec[42]	2019	Infrastructure Provisioning	Opensource	Single File, Repository Level	No

⋆ - Tools that are developed and adopted by private organisations.

Table 5.1 presents the analysis results for static code analysis tools for IaC. Among the examined tools, 30% analyse both single file and repository levels, 40% focus exclusively on repository-level analysis, and 30% perform only single file analysis. Regarding availability, 70% of surveyed tools are open-source solutions, 20% are proprietary, and 10% offer hybrid licensing models.

Commercial organisations develop 30% of these tools, while 70% originate from research environments.

The distinction between commercial and research-based tools affects their design priorities. Commercial tools prioritise production-ready environments, while research-based tools emphasise experimentation and novel approaches. Despite this diversity, none of the examined tools incorporate metadata analysis, limiting their ability to detect supply chain attacks.

5.2.3 Implications for Supply Chain Security

The absence of metadata analysis capabilities in current IaC smell detection tools means that security smells within supply chain dependencies remain undetected, resulting in vulnerable code persisting across IaC environments. Without examining dependency relationships, author information, and version histories contained in metadata, these tools cannot identify compromised dependencies or trace vulnerability propagation through dependency chains.

This limitation becomes particularly concerning given the complexity of modern IaC deployments. As repositories increasingly rely on external dependencies, the inability to analyse metadata-driven relationships leaves systems exposed to supply chain attacks that exploit trust relationships between components.

The transition from understanding tool limitations to developing solutions requires a methodology that addresses these gaps. The following section presents an approach that integrates metadata analysis with traditional static code analysis to enhance supply chain vulnerability detection.

5.3 Methodology

Building upon the identified limitations in existing tools, this section presents a methodology that incorporates metadata analysis and policy used to detect vulnerable repositories in IaC supply chains.

5.3.1 Algorithmic Approach

The methodology incorporates metadata scores and security scores from the quality assessment framework established in Section 3.3. The approach extends analysis scope from individual repositories to dependency chains through metadata examination and vulnerability propagation tracking. Algorithm 5.3.1 presents the formalisation of this approach.

Algorithm 5.3.1: Computational Methodology for Supply-Chain Attack Detection

Notation:
R: primary repository under analysis
r: repository being analysed (primary or dependency)
$M(r)$: metadata associated with repository r
$D(r)$: set of direct dependencies of repository r
$MS(r)$: metadata & $SS(r)$: security score for repository r
$V(r)$: vulnerabilities identified in repository r
S: set of repositories selected in Phase 1
$T(R)$: transitive closure of all dependencies of R
V_{total}: union of vulnerabilities across the dependency chain
θ_{MS}: metadata & θ_{SS}: security score threshold
Phase 1: Repository Selection via Metadata Score
for each repository r in dataset **do**
    Calculate metadata score $MS(r)$ with only **dependency attribute**
    if $MS(r) < \theta_{MS} = 1$ **then** Add r to selected repositories set S
Phase 2: Dependency Discovery
Initialise: $T(R) \leftarrow \emptyset$
repeat
    for each repository $r \in R \cup T(R)$ **do**
        Retrieve metadata: $M(r)$ & Extract direct dependencies: $D(r)$
        Update transitive dependency set: $T(R) \leftarrow T(R) \cup D(r)$
until $D(r) = \emptyset, \forall r \in T(R)$
Phase 3: Security Assessment via Code Security Score
for each repository $r \in R \cup T(R)$ **do** Calculate security score $SS(r)$:
    if $SS(r) < \theta_{SS} = 1$ **then**
        Flag r as vulnerable: $V(r) \leftarrow$ security smells in r
Phase 4: Vulnerability Mapping
$V_{total} \leftarrow \bigcup_{r \in R \cup T(R)} V(r)$

The algorithm executes in four phases. Phase 1 selects repositories for analysis based on metadata scores computed using the framework from Section 3.3. The metadata score calculation employs only the dependency attribute and treats it as a negative value, setting $\theta_{MS} = 1$ to identify repositories with

declared dependencies. This threshold ensures selection of repositories participating in dependency relationships.

Phase 2 performs dependency discovery through iterative metadata retrieval. For each repository r in the set $\{R\} \cup T(R)$, where R denotes the primary repository under analysis and $T(R)$ represents the transitive closure of all its dependencies, the algorithm extracts metadata $M(r)$ to identify direct dependencies $D(r)$. The process continues until no additional dependencies exist, constructing the complete set $T(R)$.

Phase 3 applies security assessment to each repository in the dependency chain. The security score $SS(r)$ calculation follows the code security category from the quality assessment framework, evaluating attributes which are deprecated keywords, hard-coded credentials, improper file permissions, logging failures, and default configurations. Repositories with $SS(r) < \theta_{SS} = 1$ are classified as containing security smells.

Phase 4 aggregates security smells across the dependency chain. The set V_{total} represents the union of all vulnerabilities $V(r)$ identified in repositories $r \in \{R\} \cup T(R)$. This aggregation enables tracking of vulnerability propagation from dependencies to dependent repositories via CWE & CVE.

The algorithm produces two outputs for each analysed repository R: a dependency tree $T(R)$ containing all transitive dependencies, and a vulnerability mapping V_{total} documenting security issues across the dependency chain. The integration of metadata scores for repository selection and security scores for vulnerability assessment demonstrates the application of the IaC code quality assessment framework to supply chain security analysis. These scores depend on the policy configuration that governs attribute weight assignments.

5.3.2 Applying Policy to Quality Computation

For supply chain vulnerability detection, the framework requires a policy configuration that targets dependency relationships and security smells. This policy activates only the Metadata and Code Security categories from the

nine-category framework, selects attributes relevant to supply chain risk, and assigns uniform importance to the selected attributes within each category. This configuration enables identification of security smell propagation paths through dependency chains.

5.3.2.1 Category Weight Configuration

This policy intent specifically targets supply chain vulnerability detection by tracking how security smells propagate through repository dependencies. Both categories receive equal priority in the assessment process, though their application serves distinct purposes within the supply chain analysis. The Metadata category functions as an initial filter, identifying repositories with external dependencies that warrant further examination. The Code Security category performs the substantive vulnerability assessment, detecting security smells that propagate through dependency relationships. This selective category application reflects the supply chain analysis requirements, where dependency presence and security vulnerabilities constitute the primary risk factors. The remaining seven categories from the framework remain inactive for this analysis, as attributes related to structure, sophistication, maintainability, functionality, error handling, automation, and integration do not directly indicate supply chain vulnerabilities.

5.3.2.2 Attribute Weight Configuration

Within the selected categories, the policy applies weighting to specific attributes that indicate supply chain risks. For metadata analysis, the policy isolates the dependencies attribute as the sole indicator, applying it as a negative attribute to determine repository inclusion in supply chain analysis. This contrasts with the best-practice policy in Section 4.5.3, where the same attribute receives a positive classification to reward integration maturity. The reclassification reflects the supply chain policy's focus on external trust relationships as risk vectors rather than modularity indicators. Repositories with

metadata scores below 1.0, indicating presence of dependencies, proceed to dependency chain traversal. The threshold ($\theta_{MS} = 1$) creates a classification boundary where repositories without dependencies bypass further analysis, while those with dependencies undergo comprehensive examination.

The security assessment phase applies equal weighting to four security attributes: hard-coded credentials, insecure file permissions, deprecated keywords, and deprecated modules. Each attribute operates as a negative attribute within the security scoring computation, treating all security smell types with equivalent severity. This approach differs from proportional weighting schemes that might prioritise certain security smell types based on exploitation difficulty or impact severity. The uniform weighting ensures that repositories containing any security smell receive scores below the threshold ($\theta_{SS} = 1$), triggering their classification as potential supply chain attack vectors. The policy configuration emphasises detection completeness over risk prioritisation, aligning with the objective of identifying all possible security smell propagation paths through dependency chains.

Under this construction, $\theta_{SS} = 1$ is not a tuned parameter but the definitional boundary of a clean repository. Because every security attribute carries negative weight, the security score equals exactly 1 if and only if no security smell is detected, and any detection reduces the score below 1. Selecting a threshold below 1 would therefore not refine the analysis but redefine it, because any $\theta_{SS} < 1$ amounts to tolerating a fixed number of detected smells.

The necessity of a threshold, rather than exhaustive smell enumeration alone, follows from the role the security score plays in the algorithm. Phase 3 selects which repositories feed the vulnerability mapping in Phase 4, and a selection step requires a decision boundary, because a list of findings alone offers no criterion for which repositories count as vulnerable when propagation is traced through a dependency chain. The threshold and the enumeration therefore serve complementary purposes. The binary flag makes triage tractable at ecosystem scale, while the detected smells $V(r)$ are retained for every flagged

repository and drive the CWE and CVE mapping in Phase 4, at the cost of severity ranking among flagged repositories. The threshold is thus not a fixed constant but the expression of a policy choice. Zero tolerance is configured here because supply chain triage rewards detection completeness, an organisation with different priorities could target other smell types through a different attribute selection, and extending the binary gate to graded severity scoring remains future work.

5.3.3 Dataset Composition

Implementation of the methodology required a dataset with accessible metadata and dependency information. As discussed in Section 4.5, Ansible Galaxy [10] served as the source due to its structured repository format containing distinct Ansible roles and collections. This structure facilitates extraction and analysis without the additional filtering required for platforms such as GitHub that contain diverse code types. Dependency information was extracted from the metadata files within each Ansible Galaxy repository and from the Ansible Galaxy API. Analysis of 31,076 repositories revealed dependency distribution patterns across the dataset. Of these, 27,801 repositories (89.5%) showed no dependencies, while 3,275 (10.5%) had identifiable dependencies.

From the 3,275 repositories with dependencies, 268 repositories and their associated dependency chains [11] were selected for analysis. The selection criterion required that each repository and its dependencies contained three metadata fields that the methodology depends on. First, dependency declarations are required by Phase 1 for repository selection and by Phase 2 for dependency chain traversal, as the algorithm extracts direct dependencies $D(r)$ from metadata $M(r)$ to construct the transitive closure $T(R)$. Second, versioning history is required for the temporal analysis presented in Section 5.4.1.2, which examines release date distributions to contextualise the prevalence of deprecated

[10] Red Hat, Inc., "Ansible Galaxy," 2016, https://galaxy.ansible.com/.

[11] Palo Alto Networks, "What Is Dependency Chain Abuse?," 2023, https://www.paloaltonetworks.com/cyberpedia/dependency-chain-abuse-cicd-sec3.

components. Third, author details are required for the author activity analysis presented in Section 5.4.1.1, which classifies repositories by maintenance status and author contribution patterns. The remaining 3,007 repositories were excluded because one or more of these three fields were absent, incomplete, or corrupted. For example, a top-level repository may declare its dependencies, yet one of those dependency repositories may lack author details, preventing the maintenance classification in Table 5.2. Similarly, a dependency repository without versioning history would prevent temporal analysis of release patterns across the chain.

Figure 5.2 illustrates example dependency chains from the dataset. The selected 268 dependency chains involved 482 total repositories, providing the foundation for analysis. The investigation focused on identifying top-level repositories vulnerable due to security issues in their dependencies rather than their own code. Top-level repositories operate exclusively as downstream consumers without serving as dependencies themselves.

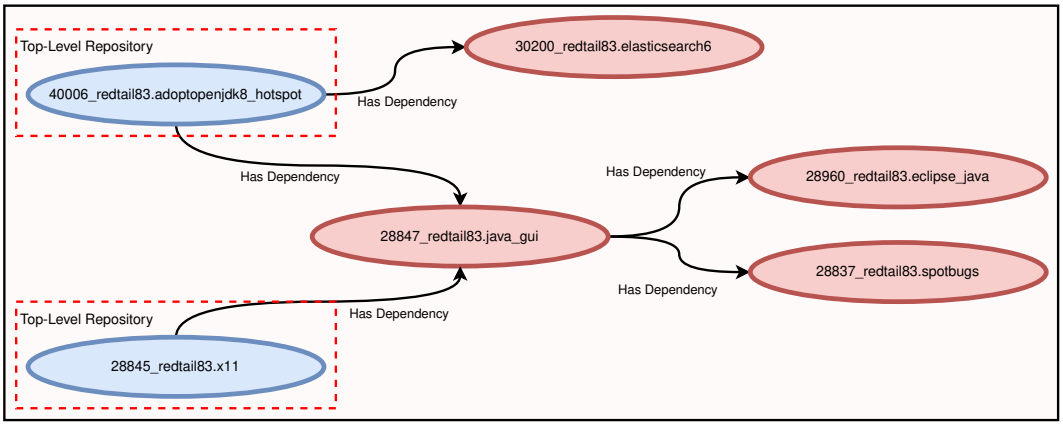


Figure 5.2: Exemplar Dependency Chains Extracted from IaC Script Dataset

The following section presents the results of the analysis and their implications for IaC security.

5.4 Evaluation and Results

Application of the metadata-assisted detection methodology to the Ansible Galaxy dataset revealed security smells within IaC dependency chains. This

section presents the findings, mapping identified security smells to established security frameworks and analysing their distribution across the ecosystem.

5.4.1 Vulnerability Distribution and Classification

Analysis following Algorithm 5.3.1 identified 45 dependency chains containing security smells that enable supply chain attacks (discussed below). These chains involved 40 repositories with security issues propagating through dependency relationships . Findings are reported at the repository level because supply chain threats arise from inter-repository dependency relationships, as discussed in Section 5.2.1, with file-level smell detections aggregated into each repository's security score . We further mapped identified security smells to the Common Weakness Enumeration (CWE) framework and known Common Vulnerabilities and Exposures (CVE) entries.

5.4.1.1 Mapping Security Smells to Common Weakness Enumeration

Security vulnerability analysis revealed predominant patterns in the types of weaknesses present in the dependency chains. The 45 dependency chains contained 40 unique repositories with security smells, as five repositories appeared as dependencies in more than one chain. Of these 40 repositories, 38 (95%) contained deprecated components, corresponding to CWE-477 (Use of Obsolete Function).[12] This reliance on outdated repositories utilising unmaintained versions of Ansible suggests that developers have not migrated their scripts to the latest versions.

The remaining two repositories contained different security smells. One repository (2.5%) contained hard-coded database credentials, corresponding to CWE-798 (Use of Hard-Coded Credentials).[13] The other repository (2.5%)

[12]MITRE, "About CWE-477: Use of Obsolete Function," 2018, https://cwe.mitre.org/data/definitions/477.html.

[13]MITRE, "About CWE-798: Use of Hard-coded Credentials," 2018, https://cwe.mitre.org/data/definitions/798.html.

exhibited improper file permissions, aligning with CWE-280 (Improper Handling of Insufficient Permissions or Privileges).[14]

5.4.1.2 Temporal Analysis of Repository Releases

Examination of repository release patterns provides context for the prevalence of deprecated components. Figure 5.3 presents the distribution of latest release dates for repositories within dependency chains. The analysis reveals a mean release date of November 2018 and a median release date of May 2018. These dates indicate that, on average, the repositories in the analysed dependency chains had not been updated for up to six years prior to the time of analysis (January 2025).

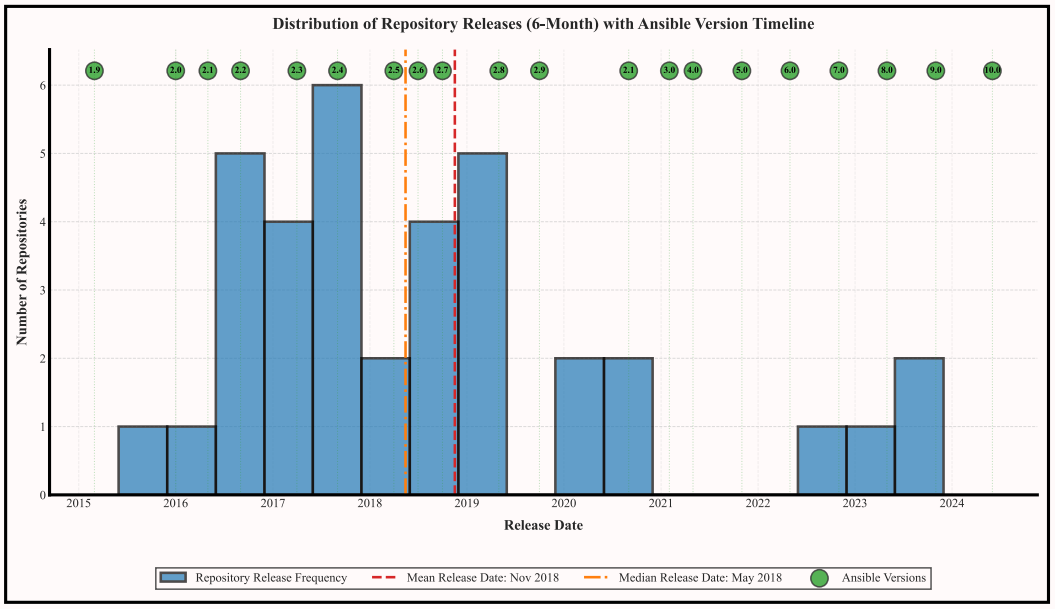


Figure 5.3: Temporal Distribution of Repository Releases with Ansible Version Release Timeline

This temporal gap correlates with the high incidence of CWE-477 weakness. As shown in Figure 5.3, each Ansible version release [15] introduces changes that deprecate previously supported keywords and modules. Repositories that have not been updated across multiple Ansible release cycles therefore contain an

[14]MITRE, "About CWE-280: Improper Handling of Insufficient Permissions or Privileges," 2018, https://cwe.mitre.org/data/definitions/280.html.

[15]Ansible Community, "Releases and Maintenance," 2025, https://docs.ansible.com/ansible/latest/reference_appendices/release_and_maintenance.html.

increasing number of deprecated components, as functionality that was valid at the time of the last update becomes obsolete with each subsequent release. These deprecated components propagate through dependency chains, creating cascading vulnerabilities in dependent systems.

5.4.2 Author Activity Patterns and Security Risks

Beyond code-level weakness, the analysis examined repository maintenance patterns through author contribution records. Repositories were classified based on three criteria. The first, repository activity, refers to whether a repository has received updates within the past six months relative to Ansible's typical release cycle. The second, author activity, indicates whether the author has contributed to any repository on the platform within the same period. The third, author type, distinguishes between repositories maintained by individual developers and those maintained by organisations. These criteria are significant because they determine the likelihood of vulnerability remediation. An inactive repository maintained by an active author presents a different risk profile from one where both the repository and the author are inactive, as the latter has no identifiable maintainer to address discovered vulnerabilities. Repository activity and author activity data were retrieved from the Ansible Galaxy API, which records the timestamps of repository updates and author contributions. Author type was determined by examining the account profiles associated with each repository on Ansible Galaxy. Table 5.2 presents the categorisation of the 40 vulnerable repositories (from the 45 dependency chains, as discussed in Section 5.4.1.1) based on these three criteria.

The categorisation reveals security risk patterns based on maintenance characteristics. Categories C1 and C2 contain repositories where the authors remain active on the platform despite the repositories themselves being inactive. C1 comprises 21 repositories maintained by individual authors with 941,327 cumulative downloads, while C2 comprises 9 repositories maintained by organisational authors with 34,370 cumulative downloads. The active sta-

Table 5.2: Repository Activity and Author Contribution Metrics in IaC Ecosystem

Category	Repository Activity	Author Activity	Author Status	Vulnerable Repositories	Cumulative Downloads
C1	inactive	active	single	21	941,327
C2	inactive	active	org	9	34,370
C3	inactive	inactive	single	5	227
C4	active	active	single	3	7,100,309
C5	inactive	inactive	org	2	72

tus of these authors suggests that remediation remains feasible if the identified risks are communicated to the maintainers.

Category C4 contains 3 repositories that are actively maintained by active individual authors, yet still contain security smells. With 7,100,309 cumulative downloads, C4 represents the highest exposure among all categories. The persistence of vulnerabilities in actively maintained repositories indicates that regular updates alone do not guarantee the resolution of security issues.

Categories C3 and C5 represent repositories that appear to be abandoned. C3 contains 5 repositories where both the repository and the individual author are inactive, while C5 contains 2 repositories with inactive organisational authors. These repositories have no identifiable maintainer to address discovered vulnerabilities. This abandonment increases their susceptibility to RepoJacking attacks [74], where malicious actors assume control of abandoned repositories to distribute compromised code to downstream dependents.

Table 5.3: Number of Ansible Repositories Affected by Supply-Chain Vulnerabilities

Category	Total Affected Repositories
C1	47
C2	19
C3	6
C4	43
C5	2

While Table 5.2 reports the 40 repositories that directly contain security

smells, Table 5.3 extends this view by counting all repositories affected through dependency relationships, including those that do not contain security smells themselves but depend on repositories that do. The total of 117 affected repositories demonstrates the propagation effect of supply chain vulnerabilities. Categories C1 and C4 account for the largest share of affected repositories (47 and 43 respectively), indicating that both inactive and actively maintained projects face risks through their dependency chains. The analysis also identified dependencies with known, documented security vulnerabilities registered in the CVE database. We discuss these findings in the following section.

5.4.3 Dependency Repositories with Identified CVEs

We identified three repositories associated with CVE entries registered in the National Vulnerability Database (NVD) among the dependency chains analysed. These cases represent the most critical supply chain risks, where known exploitable vulnerabilities propagate to dependent repositories. The findings demonstrate how metadata-assisted analysis can uncover documented vulnerabilities spreading through dependency relationships.

The first repository, maintained by an active individual author with 4,389,601 downloads, contains CVE-2024-8775 [16] with a CVSS v3.1 base score of 5.5. Despite active maintenance, the vulnerability persists, affecting numerous downstream projects through its widespread adoption.

The second repository, which is inactive but maintained by an active individual author with 1,567 downloads, also contains CVE-2024-8775. This duplication illustrates how vulnerabilities persist across different dependency chains regardless of repository popularity or maintenance status.

The third repository, managed by an inactive organisational author with 41 downloads, contains CVE-2017-7550 [17] with a CVSS v3.1 base score of 9.8.

[16]MITRE Corporation, "CVE-2024-8775: Exposure of Sensitive Information in Ansible Vault Files Due to Improper Logging," 2024, https://nvd.nist.gov/vuln/detail/CVE-2024-8775.

[17]MITRE Corporation, "CVE-2017-7550: Ansible Jenkins Plugin Module Exposes Passwords in Remote Host Logs," 2017, https://nvd.nist.gov/vuln/detail/CVE-2017-7550.

The combination of critical severity, organisational abandonment, and lack of maintenance creates conditions conducive to exploitation.

5.4.4 Responsible Disclosure and Author Feedback

Following identification of CVE-2024-8775 in a repository, we contacted the repository author through responsible disclosure. The author responded that the vulnerability pertains to Ansible's Vault functionality rather than to the repository itself, and that responsibility for security hardening depends on the user's environment and their usage of Ansible Vault. This response indicates that supply chain vulnerabilities can originate at the platform level and propagate through dependent repositories, even when the repository maintainer has not introduced any vulnerability directly.

5.4.5 Implications for IaC Security

These results demonstrate that metadata reveals security-related smell patterns within IaC environments that are difficult to identify through traditional static code analysis methods. The prevalence of deprecated dependencies with extended periods of no maintenance indicates a system-wide risk across the IaC ecosystem. The presence of known CVEs within both active and inactive repositories highlights that the maintenance status of a repository does not necessarily indicate that its security concerns have been addressed. The responsible disclosure response further demonstrates that vulnerabilities can originate at the platform level rather than within individual repositories, adding a layer of complexity to remediation efforts.

The finding that many of these vulnerabilities exist within the dependency chain rather than at the top-level repositories further emphasises the value of transitive dependency analysis when assessing the security posture of an organisation utilising IaC. Organisations evaluating their IaC environments need to assess all dependencies, including transitive dependencies, to ensure a comprehensive view of potential threats.

5.5 Discussion

The methodology clearly demonstrated its ability to identify supply chain vulnerabilities through metadata analysis. However, there are limitations which will be important to consider for future research.

5.5.1 Limitations

At present, smell detection is based on static analysis methodologies (i.e., examining code without executing it). There are likely security smells created by runtime interaction between code segments from different repositories that this approach cannot detect. Dynamic analysis performed during execution could potentially uncover other runtime security smell patterns, especially within IaC where configuration interactions influence system behaviour.

5.5.2 Future Work

Future research should focus on developing an approach combining both static and dynamic analysis for detecting smells in IaC configurations. Validation of the methodology's applicability across multiple IaC platforms beyond just Ansible would also represent an extension of the current work. In addition, development of automated tools that can perform metadata-assisted analysis to assist with adoption into production environments would represent an advancement in IaC security.

Another area of potential future research includes investigating remediation strategies to address the identified security smells. Understanding how to effectively communicate the risk associated with each identified security smell to the repository maintainers, as well as develop a process to facilitate the maintenance teams' remediation efforts would provide a pathway to reducing the window of time that vulnerabilities exist.

5.6 Chapter Summary

This chapter validated the IaC code quality framework by demonstrating its capability to detect supply chain vulnerabilities, a category of security smells not identified by existing IaC analysis tools. Application of the framework with security-focused policy configuration identified security vulnerabilities in 45 Ansible dependency chains affecting 117 repositories, confirming that quality assessment methodology extends beyond code improvement to security vulnerability detection. The framework revealed security smells through metadata analysis and dependency traversal, capabilities absent in current IaC security tools. Detected vulnerabilities mapped to established security frameworks, specifically deprecated dependencies (CWE-477), hard-coded credentials (CWE-798), and improper file permissions (CWE-280), with three repositories containing known CVEs. Temporal analysis revealed correlation between repository age and vulnerability accumulation, while author activity patterns identified exploitation risks across maintenance states. Responsible disclosure of a CVE to a repository author revealed that the vulnerability originates at the Ansible platform level, demonstrating that supply chain vulnerabilities can propagate from underlying platforms through dependent repositories. The integration of metadata attributes with quality scoring mechanisms enables identification of supply chain risks that traditional static analysis overlooks. Future development requires automated tool implementation incorporating the framework's metadata-driven approach, combined with dynamic analysis capabilities to strengthen defences against supply chain threats in IaC environments.

Chapter 6

Detecting Repository Clones and Tracking Security Smell Diffusion Patterns in IaC[1]

The preceding chapters established an IaC code quality framework with security smell detection capabilities (Chapter 5) and policy-based evaluation mechanisms (Chapter 4). The previous chapter demonstrated the framework's ability to detect supply chain vulnerabilities within dependency chains. However, dependency relationships represent only one pathway through which security smells propagate across repositories. A broader question to consider is whether the framework can be used for tracking security smell propagation through other repository relationships, particularly code duplication patterns.

Cloned repositories are entire repositories duplicated with minimal modifications such as whitespace changes, README updates, comment variations, or formatting differences. Security smell diffusion represents a distinct propagation mechanism where code patterns indicating vulnerabilities transfer from source to cloned repositories [16]. Recent incidents demonstrate the scale of

[1]This chapter largely builds upon P. R. R. Konala et al.'s research [75], incorporating revisions for thesis presentation.

this risk. GitHub identified 35,000 backdoored repository clones in 2022, [2] expanding to 100,000 by 2023, [3] and affecting one million devices by 2024.[4] As observed in Section 2.3.2, current IaC tools cannot track cross-repository relationships or compare repositories with each other, leaving practitioners unable to identify code overlap or detect when scripts derive from common sources. While platforms like GitHub maintain fork relationships through metadata, they do not detect clones created outside the forking mechanism. IaC platforms such as Ansible Galaxy lack even fork tracking mechanisms, rendering code duplication invisible and enabling security smells to propagate undetected across ecosystems. This chapter investigates security smell diffusion patterns across IaC repository clones. The investigation begins by examining clone detection techniques from traditional software engineering and their applicability to IaC contexts. With this foundation established, the analysis surveys current IaC platform capabilities and tool limitations to understand existing gaps in clone-based vulnerability detection. Building on these findings, the chapter develops a methodology that combines content and structure metrics for detecting repository clones without fork metadata, then applies this approach to real-world IaC repositories to examine what security smell diffusion patterns emerge and what implications they hold for IaC ecosystem security.

6.1 Related Work

Understanding security smell diffusion in IaC requires examining code duplication concepts and detection techniques from traditional software engineering. This section establishes foundational knowledge about clone detection methods

[2] Ax Sharma, "35,000 Code Repos Not Hacked—But Clones Flood GitHub to Serve Malware," BleepingComputer, August 2022, https://www.bleepingcomputer.com/news/security/35-000-code-repos-not-hacked-but-clones-flood-github-to-serve-malware/.

[3] Matan Giladi and Gil David, "Over 100,000 Infected Repos Found on GitHub," Apiiro Blog, 28 February 2024, https://apiiro.com/blog/malicious-code-campaign-github-repo-confusion-attack/.

[4] Microsoft Threat Intelligence, "Malvertising Campaign Leads to Info Stealers Hosted on GitHub," 2025, https://www.microsoft.com/en-us/security/blog/2025/03/06/malvertising-campaign-leads-to-info-stealers-hosted-on-github/.

and their application to infrastructure management contexts.

6.1.1 Clone Detection in General-Purpose Programming Languages

Code clones represent duplicated fragments sharing similarity through exact replication, syntactic variations, or functional equivalence [76]. These duplications arise from copy-paste practices, design pattern implementation, API constraints, or developer habits. Clone detection remains essential for software maintenance as duplicates affect system evolution and facilitate bug propagation.

Clone detection methods have evolved through three decades into five categories. Text-based approaches compare code as line sequences through hashing techniques. Johnson [77, 78] pioneered these methods, later enhanced by tools such as sif [79] and Duploc [80]. Token-based detection emerged with Baker [81] introducing parametrised suffix trees, advanced by tools such as CCFinder [82] and CP-Miner [83] for scalability. Mayrand et al. [84] developed metric-based approaches comparing numerical code representations.

Structural analysis techniques emerged with CloneDR [85] introducing AST-based methods using sub-tree hashing. DECKARD [86] enhanced this approach with locality-sensitive hashing. Hybrid approaches such as NiCad [87] combine normalisation with sequence matching. Roy et al. [88] later synthesised these tools and techniques, defining clone types and providing benchmark evaluations.

6.1.2 Clone Detection in IaC

The transition from traditional software to IaC contexts reveals domain-specific clone detection requirements. IaC scripts exhibit patterns through template copying and configuration reuse that differ from traditional software.

Tsuru et al. [89] analysed 5,000 Dockerfiles using Type-2 clone detection,

separating Docker syntax from shell scripts. Their domain-specific analyser achieved 95% precision through file-by-file analysis. Oumaziz et al. [90] investigated Dockerfile reuse patterns, identifying three management strategies using index-based clone detection, template-based generation, and internal tool development.

Security implications of IaC cloning patterns emerge through empirical studies. Rahman et al. [91] demonstrated insecure snippet propagation to 35 infrastructure resources, with 92% of anti-patterns replicating across Puppet [5] scripts. Additionally, Dean et al. [92] found that approximately 99% of insecure anti-pattern code snippets propagate from original repositories to their forks in the Open-Source Software (OSS) ecosystem.

Existing IaC clone detection research focuses on individual files rather than complete repositories. This approach misses repository-wide patterns and interdependencies. IaC repositories contain various resource and configuration file types including README files, templates, YAML configurations, and shell scripts that collectively define infrastructure. Security smell diffusion occurs across these interconnected files within codebases. Therefore, detection approaches must analyse complete repositories rather than isolated scripts.

6.1.3 Repository Metadata Schema

The analysis of security smell diffusion requires understanding IaC platform metadata structures. Section 2.1.2 established the three IaC categories and platform-specific terminology. This subsection examines the metadata these platforms provide for tracking repository relationships.

Listing 6.1.3 presents metadata available through Ansible Galaxy's API. The metadata includes dependencies, version identifiers, namespace associations, download statistics, authorship information, and repository sources. Ansible repositories can link to GitHub through metadata parameters `github_repo`, `github_branch`, and `github_user` (highlighted in lines 2–4). These parame-

[5] Puppet, Inc., "Puppet," 2023, https://puppet.com/.

ters enable access to extended GitHub metadata including fork-chain details, commit histories, and version control. We, however, found that the majority of Ansible Galaxy repositories do not populate these GitHub linkage parameters, resulting in limited GitHub integration across the ecosystem. Without this integration, our analysis cannot rely on fork metadata to trace code origins and track security smell diffusion, requiring alternative methods for identifying repository relationships.

Listing 6.1.3: Sample Ansible Galaxy Repository Metadata

```
1: { "id": 56789,"created": "2024-02-05","username": "devops_admin",
2:   "github_repo": "ansible-role-nginx",
3:   "github_user": "devops_admin",
4:   "github_branch": "master",
5:   "name": "nginx","summary_fields": {
6:     "dependencies": [{"id": 10,"name": "sec.hardening"},"..."],
7:     "namespace": {"id": 3050,"name": "devops_admin"},
8:     "provider_namespace": {"repository": {"name": "nginx"},
9:     "tags": ["webserver","nginx", "..."],
10:    "versions": [{"name":"1.1"}, "..."]}},"downloads": 342560}
```

6.2 IaC Tool Capabilities for Diffusion Analysis

The investigation of security smell diffusion requires understanding the current landscape of IaC platforms and analysis tools. This section examines platform capabilities and tool limitations to establish the context for developing clone detection methodologies.

6.2.1 Survey of IaC Platform Characteristics

The survey of IaC platforms reveals that repositories and metadata vary in availability across different categories. Among 10 examined platforms, 5 maintain repositories with searchable components for sharing reusable infrastructure definitions.

Analysis of platform characteristics shows that Terraform Registry, Ansible Galaxy, Puppet Forge, Chef Supermarket, and Docker Hub provide APIs

for programmatic access. These platforms expose 15–25 metadata attributes per artifact including version information, dependencies, download statistics, and trust relationships. Four platforms integrate with GitHub as an external source, while Docker Hub operates without external integration. The remaining 5 platforms (CloudFormation, Azure Resource Manager Templates, Google Cloud Deployment Manager, SaltStack, and Packer) rely on GitHub for code distribution without platform-specific codebases. Table 6.1 summarises the findings.

Table 6.1: Data Availability and Integration Features of IaC Distribution Platforms

IaC Category	IaC Platform	IaC Codebase	API Access	Metadata Parameter	Fork Metadata	External Integration
Infrastructure Provisioning	Terraform	Terraform Registry	✓	19	✗	✓
Configuration Management	Ansible	Ansible Galaxy	✓	20	✗	✓
	Puppet	Puppet Forge	✓	25	✗	✓
	Chef	Chef Supermarket	✓	15	✗	✓
Image Building	Docker	Docker Hub	✓	25	✗	✗

Legend: ✗ Not Available, ✓ Available

The combination of structured metadata, API accessibility, and external GitHub integration provides the infrastructure necessary for analysing repository clones across IaC artifacts. These capabilities enable detection of duplication patterns and security vulnerabilities through code reuse analysis. Whether existing IaC security tools exploit these capabilities for clone-based vulnerability detection remains an open question.

6.2.2 IaC Security Tool Capabilities Analysis

To address this question, the IaC smell detection tools surveyed in Section 2.3.2 were systematically analysed for clone detection capabilities. The survey examined 24 tools spanning open-source, proprietary, academic research, and commercial solutions. These tools, developed between 2011 and 2023, detect code and security smells across infrastructure provisioning, configuration management, and image building categories.

We found two limitations across all examined tools. First, as established in Section 5.2.2, none provide metadata analysis functionality. Second, we found that despite accepting input as either single files or complete repositories, none of the surveyed 24 state-of-the-art tools support file-to-file or repository-to-repository comparison functionality for IaC scripts. This limitation prevents analysis of security smell diffusion from original to cloned repositories, making it hard to track vulnerability propagation through code reuse.

While platforms provide metadata through APIs with 15-25 attributes per artifact and 4 of 5 dedicated platforms integrate GitHub for cross-platform analysis, none of the 24 IaC analysis tools support metadata analysis or cross-repository comparison. This disconnect motivates the development of techniques leveraging metadata for repository clone detection and security smell tracking.

6.3 Methodology

In this section, we present two algorithms for detecting repository clones and tracking security smell diffusion. The algorithms identify repository clones, which are entire repositories duplicated with minimal modifications such as whitespace changes, README updates, comment variations, or formatting differences. For such minimally modified clones, content and structure metrics effectively capture similarities, as the changes are superficial rather than semantic. We focus on this approach because IaC codebases lack clone detection tools as established in subsection 6.2.1. The first algorithm calculates similarity between repository pairs to identify cloning relationships, while the second tracks security smell diffusion using these similarity scores. More sophisticated detection involving semantic analysis for heavily modified repositories remains beyond our current scope.

6.3.1 Computing Repository Similarity Metrics

Algorithm 6.3.1 analyses repository collections to determine relationships between them. Given a set of repositories ($\mathcal{R}$), it compares every unique pair ($\mathcal{P}$) once, avoiding redundant calculations. Each pair (r_i, r_j) is evaluated through three metrics ($S_{content}$, $S_{structure}$, $S_{overall}$) capturing different relationship aspects. The algorithm's time complexity is $O(n^2)$, where n is the number of repositories in the collection $\mathcal{R}$, as it must process $\binom{n}{2} = \frac{n(n-1)}{2}$ unique pairs. The space complexity is also $O(n^2)$ due to storing all pairwise similarity scores.

Algorithm 6.3.1: Computation of IaC Script Similarity for Diffusion Pattern Analysis

Input: Set of repositories $\mathcal{R} = \{r_1, r_2, ..., r_n\}$
Output: Similarity values $S_{ij} = \{S_{overall}\}$ for each pair (r_i, r_j)
Notation:
    r_i, r_j – individual repositories, $\mathcal{F}(r)$ – set of files in repository r
    $\mathcal{L}(r)$ – total lines in repository r, $\Delta(r_i, r_j)$ – git-diff statistics
Pairwise Comparison:
    Generate pairs: $\mathcal{P} \leftarrow \{(r_i, r_j) : r_i, r_j \in \mathcal{R}, i < j\}$

Similarity Computation:
for all $(r_i, r_j) \in \mathcal{P}$ **do**
    Compute similarity: $S_{ij} \leftarrow \text{ComputeSimilarity}(r_i, r_j)$
ComputeSimilarity(r_i, r_j)**:**
Step 1 Content Similarity:
    Get files: $\mathcal{F}_i \leftarrow \text{GetFiles}(r_i)$, $\mathcal{F}_j \leftarrow \text{GetFiles}(r_j)$
    Count lines: $\mathcal{L}_i \leftarrow \sum_{f \in \mathcal{F}_i} \text{CountLines}(f)$,
        $\mathcal{L}_j \leftarrow \sum_{f \in \mathcal{F}_j} \text{CountLines}(f)$
    Total lines: $\mathcal{L}_{total} \leftarrow \mathcal{L}_i + \mathcal{L}_j$
    Get diff: $\Delta(r_i, r_j) \leftarrow \text{GitDiff}(r_i, r_j)$
    Changed lines: $\mathcal{L}_{changed} \leftarrow \sum_{d \in \Delta}(d.additions + d.deletions)$
    $S_{content} \leftarrow 100 \times \max(0, 1 - \mathcal{L}_{changed}/\mathcal{L}_{total})$
Step 2 Structure Similarity:
    Get paths: $\mathcal{P}_i \leftarrow \text{RelativePaths}(\mathcal{F}_i)$, $\mathcal{P}_j \leftarrow \text{RelativePaths}(\mathcal{F}_j)$
    Jaccard similarity: $S_{structure} \leftarrow 100 \times |\mathcal{P}_i \cap \mathcal{P}_j| / |\mathcal{P}_i \cup \mathcal{P}_j|$
Step 3 Overall Similarity:
    return $S_{overall} = (S_{content} + S_{structure})/2$

In IaC contexts, both content and structural analysis provide comprehensive similarity assessment, though structure plays a more vital role than in traditional software development. While traditional software allows flexible file organisation, IaC frameworks enforce specific directory structures that directly map to infrastructure components. For instance, Ansible roles require precise placement of tasks, handlers, and variables in designated directories. Content similarity ($S_{content}$) reveals code sharing degree, identifying repositories with similar infrastructure configurations. Structure similarity ($S_{structure}$) becomes particularly significant as directory hierarchies in IaC represent deployment architectures and operational workflows, not merely code organisation preferences. Combined, these metrics distinguish repositories sharing superficial structures from those containing substantially similar infrastructure definitions, with structural patterns often indicating whether repositories target the same deployment scenarios. Below we discuss how the different similarity scores are calculated.

Content similarity ($S_{content}$) examines code differences between repositories using `git-diff` [6]. We selected `git-diff` for its widespread adoption in handling large-scale version control systems [93] and its ability to detect line-level changes across text-based files. The following `git-diff` options were used for repository comparison. `--no-index` enables comparison of directories without requiring git initialisation, which is necessary because downloaded Ansible Galaxy repositories are not git repositories. `--numstat` provides machine-readable output of additions and deletions per file, allowing automated processing of results across repository pairs. `--ignore-all-space` and `--ignore-blank-lines` normalise formatting variations that are common in IaC files, ensuring that whitespace differences do not inflate dissimilarity scores. `--find-renames=90%` matches files whose content is at least 90% similar under different filenames, addressing the case where a repository clone renames a file while leaving its content largely unchanged. A pure rename

[6] Git SCM, "git-diff," 2025, https://git-scm.com/docs/git-diff.

therefore costs no content similarity, and a rename combined with minor edits counts only the edited lines. The 90% threshold is deliberately stricter than the git default of 50%, because pairing genuinely different files that happen to share half their lines would inflate similarity between unrelated repositories, whereas requiring exact equality would decompose a rename carrying even one trivial edit into a full deletion and addition, understating similarity for a true clone. The algorithm counts total lines ($\mathcal{L}_{total}$) across all text files in both repositories (r_i, r_j), identifies changes through git diff ($\Delta(r_i, r_j)$), and calculates similarity as the percentage of unchanged content. The total line count combines lines from both repositories, meaning that differences in repository size are reflected in $\mathcal{L}_{total}$. If, for example, two repositories contain 600 and 400 lines respectively, $\mathcal{L}_{total}$ equals 1000. If 200 lines differ between them, the content similarity equals 80%.

Structure similarity ($S_{structure}$) analyses repository organisation using the Jaccard coefficient [94] to compare file path sets. The Jaccard coefficient was chosen as it provides a normalised measure of set overlap that effectively captures structural correspondence between repositories regardless of their absolute sizes [95], making it ideal for comparing projects of varying scales. The algorithm extracts relative file paths ($\mathcal{P}_i$, $\mathcal{P}_j$) and measures overlap. For example, if two repositories share 30 paths among 50 unique paths total, the structure similarity equals 60%.

Overall similarity ($S_{overall}$) combines content and structure measurements through arithmetic mean, reflecting equal contribution of code content and organisational structure to repository relationships.

A renamed file is treated differently by the two components. Content similarity pairs it with its counterpart through rename detection, so the rename itself costs nothing, yet structure similarity penalises it because the renamed path leaves the intersection of the two path sets. A clone that renames many files could therefore fall below the similarity threshold and escape detection, which affects recall but never the precision of the pairs that are flagged. Mea-

surement across the 109 known GitHub fork pairs used as ground truth shows this risk to be small in practice. Renamed files appeared in 3 pairs (3.61%), accounting for 11 of 988 file comparisons (1.11%), and 10 of the 11 were pure renames with identical content, matched at zero content-similarity cost. The single edited rename (`LICENSE.md` to `LICENSE`) was still paired correctly at the 90% threshold. No renamed files appeared in the Ansible Galaxy pairs above the threshold, and no pair in the dataset combined content similarity above 90% with an overall similarity below the threshold, confirming that no near-clone was lost to renaming.

6.3.2 Computing Security Smell Diffusion Patterns

Algorithm 6.3.2 applies overall similarity scores to identify security smells and track their diffusion patterns. Since platforms such as Ansible Galaxy lack explicit fork information, the algorithm infers cloning relationships based on similarity percentages. The methodology applies a similarity threshold (τ) to filter repository pairs ($\mathcal{P}$), considering only those exceeding the threshold as potential repository clones ($\mathcal{Q}$). Only repository pairs in the set $\mathcal{Q}$ undergo further analysis for security smell diffusion.

Following identification of potential repository clones, the algorithm determines source-clone relationships using 'creation timestamps (t_i, t_j)' from repository metadata. The earlier-created repository is designated as the source and the later one as the cloned repository, establishing the direction of code flow and security smell diffusion. While temporal ordering may not capture all edge cases (such as simultaneous development), IaC platform-provided timestamps serve as a reliable proxy for source-clone relationships in the vast majority of non-malicious code reuse scenarios.

Algorithm 6.3.2: Computation of Security Scores for Smell Detection in IaC Scripts

Input: Repository pairs with overall similarity scores $\mathcal{P} = \{(r_i, r_j, S_{ij})\}$,
threshold (τ)
Output: Security smell diffusion map $\mathcal{V}$
Cloned Repository Detection:
$\mathcal{Q} \leftarrow \{(r_i, r_j, S_{ij}) \in \mathcal{P} : S_{ij}.overall \geq \tau\}$
Source Identification & Analysis:
for all $(r_i, r_j, S_{ij}) \in \mathcal{Q}$ **do**
    $t_i, t_j \leftarrow \text{GetCreationTime}(r_i), \text{GetCreationTime}(r_j)$
    $(source, clone) \leftarrow (r_i, r_j)$ if $t_i < t_j$ else (r_j, r_i)
    AnalyseSecuritySmells$(source, clone)$
AnalyseSecuritySmells$(source, clone)$**:**
$\mathcal{C}_{source} \leftarrow \text{MapCWEs}(\text{ScanSecuritySmells}(source))$
$\mathcal{C}_{clone} \leftarrow \text{MapCWEs}(\text{ScanSecuritySmells}(clone))$
$\sigma_{SS}^{source} \leftarrow \text{ComputeSecurityScore}(\mathcal{C}_{source})$
$\sigma_{SS}^{clone} \leftarrow \text{ComputeSecurityScore}(\mathcal{C}_{clone})$
if $\sigma_{SS}^{clone} = \sigma_{SS}^{source}$ **then**
    $label \leftarrow$ Propagated
else if $\sigma_{SS}^{clone} < \sigma_{SS}^{source}$ **then**
    $label \leftarrow$ Induced
else
    $label \leftarrow$ Fixed
$\mathcal{V} \leftarrow \mathcal{V} \cup \{(source, clone, label, \mathcal{C}_{source}, \mathcal{C}_{clone})\}$

Both repositories undergo attribute-based static code analysis following the methodology presented in Section 3.3, which employs regular expressions to identify security smells. The detected smells are mapped to CWEs [7] and assigned security scores , denoted σ_{SS} to distinguish the score itself from the selection threshold θ_{SS} used in Chapter 5 . The security score is computed from a baseline value of 1, where each detected occurrence of a security smell reduces the score. Repositories with no security smells retain a score of 1, while repositories with security smells score below 1. Since the score decreases proportionally with the number of detected occurrences, it enables comparison of security characteristics between source and clone repositories at a granular

[7] MITRE Corporation, "Common Weakness Enumeration," 2025, https://cwe.mitre.org/index.html.

level rather than through binary presence or absence.

The algorithm assigns each repository pair to one of three diffusion categories through comparative analysis of the aggregate security scores of source and clone :

- **Propagated** : $\sigma_{SS}^{clone} = \sigma_{SS}^{source}$, indicating a security posture that persists unchanged through the cloning process
- **Induced** : $\sigma_{SS}^{clone} < \sigma_{SS}^{source}$, representing cases where the clone exhibits more security smells than the source
- **Fixed** : $\sigma_{SS}^{clone} > \sigma_{SS}^{source}$, indicating cases where the clone demonstrates fewer security smells than the source

This score-based categorisation provides granular insights into security smell transmission patterns across the ecosystem. The approach identifies not only which security smells persist through code reuse but also quantifies the severity changes during propagation. The resulting diffusion map ($\mathcal{V}$) captures the security evolution from source to clone repositories together with the per-CWE profiles of both sides , enabling identification of repositories that improve security postures and those that introduce security degradation.

Because each repository pair receives one net label, mixed changes could in principle offset each other in the aggregate score. A clone that removes one vulnerability while introducing another of equal weight would be labelled propagated despite a changed security profile. We measured the prevalence of this netting scenario by comparing the per-CWE profiles of source and clone within every classified pair. Across all 1,008 pairs, only 2 of the 863 propagated pairs (0.23%) show differing per-CWE profiles despite equal scores, 2 of the 94 fixed pairs contain a masked induction, and 1 of the 51 induced pairs contains a masked fix. In every pair the assigned label agrees with the sign of the score difference. Score equality is therefore a reliable proxy for an unchanged security state in this dataset, and the direction labels are stable against mixed changes. Substitution within a single CWE category at an identical occurrence

count remains invisible to any count-based analysis, and mapping individual vulnerability instances from source to clone would make such cases directly observable, as noted under future work. These security score computations require explicit policy configuration to determine attribute weights.

6.3.3 Applying Policy to Quality Computation

For security smell diffusion analysis, the framework requires a policy configuration that targets security smell propagation across repository clones. This policy activates only the Code Security category from the nine-category framework, selects attributes relevant to security smell detection, and assigns uniform importance to the selected attributes. This configuration enables comparison of security vulnerabilities across source and cloned repositories without bias toward specific security smell types.

6.3.3.1 Category Weight Configuration

This policy configuration tracks security smell propagation patterns across repositories by focusing the assessment on security vulnerabilities that spread through repository clones. The Code Security category receives full weight in the assessment, while the remaining eight categories (Metadata, Code Structure, Code Sophistication, Code Maintainability, Functionality & Purpose, Error Handling, Code Automation, and Code Integration) are assigned a weight of zero. Setting these categories to zero ensures that the quality computation reflects only security smell occurrences, isolating the security dimension for diffusion analysis.

6.3.3.2 Attribute Weight Configuration

For security smell diffusion analysis, the methodology applies equal weighting to four negative attributes ($\tau_j = -1$) from the Code Security category. Hard-coded credentials represent security smells through password-related strings embedded in IaC scripts that create exploitation vectors in supply chain at-

tacks. Insecure file permissions manifest as configuration settings that permit unauthorised access or modification of infrastructure components. Deprecated keywords and modules indicate usage of outdated Ansible directives containing known security vulnerabilities that led to deprecation. Suspicious comments reveal workaround solutions, TODO markers, or security-related technical debt that may indicate incomplete security implementations.

Each occurrence reduces the security score from the baseline value of 1, resulting in repositories with security smells scoring below 1. This scoring approach (1 = no security smells, <1 = security smells present) provides a classification mechanism for identifying repositories requiring further analysis. The equal weighting ensures that each security smell type contributes equally to the score reduction, preventing the diffusion analysis from being skewed toward any single smell type.

6.3.4 Dataset Composition

The Ansible Galaxy dataset described in Section 4.5 provides the empirical foundation for this analysis. We selected three technologies representing different infrastructure categories. Elasticsearch (search engine, 188 repositories), Jenkins (CI/CD platform, 210 repositories), and MySQL (database, 491 repositories), totalling 889 unique repositories from Ansible Galaxy, excluding commit versions of the same repository in these counts. These technologies were chosen for their diverse repository counts, widespread adoption in production environments, and representation of critical infrastructure components where security vulnerabilities have an impact. Pairwise comparison generated unique pairs: 17,578 for Elasticsearch, 21,945 for Jenkins, and 120,295 for MySQL, providing sufficient data to assess cloning patterns while maintaining computational feasibility for initial validation.

6.3.5 Validation via Known Fork Metadata Relationship

We chose to use repositories from Ansible Galaxy that link to GitHub's fork chain data as ground truth for methodology validation, because Ansible Galaxy and other IaC codebases lack repository fork information. This enabled us to leverage GitHub's fork chain information for calculating accuracy, precision, recall, and F1 score metrics. As described in Section 6.2.1, repository authors in Ansible Galaxy can link their repositories to GitHub, which provides fork chain information through the `github_repo`, `github_branch`, and `github_user` metadata attributes.

The validation process analysed metadata from three repository technologies to identify GitHub linkages through these attributes. They function as metadata indicators where their presence confirms GitHub integration. Repositories containing all three attributes achieve a metadata score of 1, indicating complete GitHub linkage that provides access to fork chain information.

This analysis identified documented GitHub fork relationships. These consisted of 43 pairs (37 unique repositories) from Elasticsearch, 34 pairs (30 unique repositories) from MySQL, and 32 pairs (33 unique repositories) from Jenkins. For each relationship, we retrieved the corresponding GitHub commit version repositories at fork creation time, enabling temporal alignment for $S_{overall}$ calculations. The known fork pairs exhibited high $S_{overall}$ values across all technologies. Elasticsearch had a mean of 81.02% and median of 89.60%, Jenkins had a mean of 86.16% and median of 93.56%, and MySQL had a mean of 89.01% and median of 95.22%. Based on the mean and median values across all three technologies exceeding 80%, we established our similarity threshold $\tau = 80\%$.

To determine whether this threshold correctly distinguishes fork pairs from non-fork pairs, we constructed a test set for each technology. Each test set consists of the repositories involved in known fork relationships, including their

commit versions at fork creation time, augmented by 10 randomly selected non-fork repositories from Ansible Galaxy. For Elasticsearch, the test set contains 79 fork chain repositories and 10 non-fork repositories, totalling 89 repositories and $\binom{89}{2} = 3{,}916$ pairs. For Jenkins and MySQL, the test sets each contain 62 fork chain repositories and 10 non-fork repositories, totalling 72 repositories and $\binom{72}{2} = 2{,}556$ pairs. Of these pairs, the known fork pairs (43 for Elasticsearch, 32 for Jenkins, and 34 for MySQL) represent the positive cases. The remaining pairs, which include combinations among the non-fork repositories, combinations between non-fork and fork chain repositories, and fork chain repository pairs without a documented fork relationship, represent the negative cases that should score below τ. We applied τ to classify all pairs in each test set as either fork or non-fork and compared these classifications against the known ground truth. Table 6.2 presents the validation results.

Table 6.2: Confusion Matrix and Performance Metrics for Pairwise Fork Detection Across Test Sets of Elasticsearch (3,916 pairs), Jenkins (2,556 pairs), and MySQL (2,556 pairs)

Technology	Confusion Matrix	Predicted Fork	Predicted Not Fork	Accuracy	Precision	Recall	F1 Score
Elasticsearch	**Actual Fork**	30 (TP)	13 (FN)	**99.67%**	**100%**	**69.77%**	**82.19%**
	Actual Not Fork	0 (FP)	3,873 (TN)				
Jenkins	**Actual Fork**	29 (TP)	3 (FN)	**99.88%**	**100%**	**90.62%**	**95.08%**
	Actual Not Fork	0 (FP)	2,524 (TN)				
MySQL	**Actual Fork**	28 (TP)	6 (FN)	**99.77%**	**100%**	**82.35%**	**90.32%**
	Actual Not Fork	0 (FP)	2,522 (TN)				

In our validation methodology context, True Positives (TP) represent actual forks correctly identified by the algorithm. False Negatives (FN) are actual forks misclassified as non-forks due to modifications reducing their $S_{overall}$ below τ. False Positives (FP) would indicate non-fork pairs misidentified as forks, but none occurred since all non-fork repository pairs fell below τ. True Negatives (TN) represent non-fork pairs accurately classified, confirmed by their $S_{overall}$ values remaining below τ. The algorithm achieved accuracy of 99.67%–99.88% across all three technologies with 100% precision since no false positives occurred.

The false negatives (13 in Elasticsearch, 3 in Jenkins, 6 in MySQL) repre-

sent forks whose $S_{overall}$ fell below τ due to modifications. Though technically forks, they evolved substantially enough to constitute new repositories. When a fork undergoes substantial modification, the code changes are likely to alter the security characteristics of the repository. Security smells present in the source may be removed through refactoring, and new security smells may be introduced through added code. As a result, the security smell profile of a heavily modified fork diverges from that of its source, reducing the relevance of tracking diffusion between the two [96]. If such forks are included in the diffusion analysis, security smells detected in them may be incorrectly attributed to the cloning process rather than to the independent changes made by the fork author. For instance, if a fork replaces the original Ansible tasks with a rewritten implementation, any security smells in the rewritten code originate from the fork author's decisions rather than from the source repository. Attributing these smells to diffusion would produce misleading results. Therefore, classifying these pairs as non-forks ensures that the diffusion analysis focuses on repository pairs where code overlap is sufficient for security smells to transfer from source to clone.

6.3.6 Contextualisation Against Existing Work

As established in Section 6.2.2, the 24 IaC smell detection tools analysed detect security smells within individual repositories but lack capabilities to analyse metadata, identify repository clones, or track security smell diffusion across IaC codebases. Direct comparison with traditional software code clone detection tools is not feasible as they operate on fundamentally different code structures than infrastructure configuration scripts. The closest available reference point is Tsuru *et al.* [89], who analysed 5,000 stand-alone Docker scripts and achieved 95% precision. However, their study reports only precision without providing accuracy, recall, or F1 scores, limiting the extent of comparison. Their work also focuses on image building rather than configuration management, and configuration management scripts exhibit different structural and

reuse patterns compared to container build files.

6.4 Evaluation and Results

This section presents the results of our methodology and examines security smell diffusion in Ansible Galaxy repositories. We analyse the complete Ansible Galaxy ecosystem, tracking how CWEs and CVEs [8] propagate through identified repository clones and their relationships.

6.4.1 Repository Clone Detection in Ansible Galaxy

After validating our methodology using known fork metadata, we applied it to the full repository sets including commit versions. The inclusion of commit versions increased the repository counts from 188 to 226 for Elasticsearch, from 210 to 240 for Jenkins, and maintained 491 for MySQL, producing 25,425 pairs for Elasticsearch, 28,680 pairs for Jenkins, and 120,295 pairs for MySQL. We applied the similarity threshold (τ) established during validation to identify repository clones. The analysis found that 87 repositories (38.4%) out of 226 Elasticsearch repositories formed 154 similarity pairs, 130 repositories (54.1%) out of 240 Jenkins repositories formed 473 pairs, and 190 repositories (38.7%) out of 491 MySQL repositories formed 3,539 pairs. These findings indicate that 38.4%–54.1% of repositories across the three analysed technologies share code overlap with at least one other repository. This level of code reuse across Elasticsearch, Jenkins, and MySQL creates pathways for security smell diffusion across multiple repositories. This is particularly concerning because, unlike GitHub which maintains fork metadata, Ansible Galaxy provides no mechanism for identifying repository clones or tracing code origins within its platform.

[8] MITRE Corporation, "Common Vulnerabilities and Exposures (CVE)," 2025, https://cve.mitre.org/.

6.4.2 Mapping Security Smells to CWE

Our analysis of security smell diffusion involved mapping detected security smells to CWEs and studying their patterns in cloned repositories. Initial scanning revealed prevalent security smells, namely suspicious comments exposing sensitive information, hard-coded credentials, and obsolete functions. We mapped these findings to three CWEs that commonly affect IaC. CWE-546 [9] manifests through suspicious comments revealing sensitive system information, internal configurations, or debugging data. CWE-798 [10] occurs when passwords, tokens, or API keys are embedded directly in code. CWE-477 [11] involves deprecated modules, plug-ins, or outdated syntax containing known security smells. Table 6.3 presents the diffusion patterns for Ansible Galaxy repositories with GitHub fork metadata (Source-Fork Pairs) and Ansible Galaxy repositories identified as clones without fork metadata linkage (Source-Clone Pairs).

The analysis reveals distinct security smell patterns across technologies. CWE-477 (obsolete modules & plug-ins) and CWE-546 (suspicious comments) show the highest propagation rates across all three technologies, while CWE-798 (hardcoded credentials) appears minimally. The prevalence of CWE-477 aligns with findings in Section 5.4.1.2 that Ansible Galaxy repositories have a mean release date of November 2018 and median of May 2018, representing approximately seven years without updates at the time of our study (July 2025). This temporal gap explains the widespread use of deprecated modules and plug-ins. Induced diffusion patterns, where repository clones introduce new security smells occur frequently. Elasticsearch repositories primarily exhibit CWE-477 and CWE-546, MySQL shows CWE-798 occurrences, and Jenkins demonstrates limited CWE-477 instances. Fixed diffusion patterns reveal a

[9] MITRE Corporation, "CWE-546: Suspicious Comments," 2025, https://cwe.mitre.org/data/definitions/546.html.

[10] MITRE, "About CWE-798: Use of Hard-coded Credentials," 2018, https://cwe.mitre.org/data/definitions/798.html.

[11] MITRE, "About CWE-477: Use of Obsolete Function," 2018, https://cwe.mitre.org/data/definitions/477.html.

Table 6.3: Analysis of Security Smell Diffusion Patterns Using Cumulative Common Weakness Enumeration Counts

Technology	Diffusion Patterns	Propagated			Induced			Fixed		
	CWE ID	546	798	477	546	798	477	546	798	477
Elasticsearch	Source	22	0	28	0	0	1	2	0	0
	Fork	22 ↕	0	28 ↕	1 ↑	0	2 ↑	0 ↓	0	0
	Source	15	6	31	6	0	0	4	0	9
	Clone	15 ↕	6 ↕	31 ↕	6 ↕	0	5 ↑	7 ↑	0	0 ↓
Jenkins	Source	1	0	15	0	0	0	10	0	13
	Fork	1 ↕	0	15 ↕	0	0	0	3 ↓	0	0 ↓
	Source	14	0	36	0	0	0	4	0	8
	Clone	14 ↕	0	36 ↕	0	0	2 ↑	0 ↓	0	0 ↓
MySQL	Source	0	0	2	0	0	0	0	0	2
	Fork	0	0	2 ↕	1 ↑	0	3 ↑	0	0	0 ↓
	Source	1	2	4	0	0	0	2	16	11
	Clone	1 ↕	2 ↕	4 ↕	2 ↑	7 ↑	16 ↑	0 ↓	0 ↓	0 ↓

Note: The values represent total CWE instances across repository pairs in specific diffusion pattern category. Arrow indicators show changes from source to fork/clone: ↕ same as source; ↑ increase; ↓ decrease in CWE counts.

concerning trade-off in Elasticsearch and Jenkins. Developers remove obsolete functions, decreasing CWE-477, but introduce suspicious comments containing quick fixes and workarounds, increasing CWE-546. This substitution suggests developers address deprecated functions by embedding temporary solutions in comments, inadvertently creating new information exposure security smells while attempting to modernise outdated code.

6.4.3 Author Practices and User Selection Patterns

The diffusion patterns reveal three observable author practices.

- Repositories are cloned without modification from sources containing security smells, perpetuating existing smells.
- Some repositories show security smell remediation during the cloning process.
- New security smells appear in cloned repositories that were absent in the source.

These induced security smells may result from various factors including coding errors, dependency changes, or configuration modifications during the cloning process. Such security smells pose threats because users may select repository clones believing them to be improved versions. This security gap in Ansible Galaxy becomes apparent when comparing platforms. Repositories with GitHub's fork chain metadata enable tracking of smell origins, whereas Ansible Galaxy lacks such mechanisms despite containing identical security smells.

Table 6.4: Distribution of Cumulative Repository Downloads by Security Smell Diffusion Pattern Category

Technology	Relationship	Propagated	Induced	Fixed
Elasticsearch	Source	4,865,481	797,204	5
	Fork	5,079	227	5,962
	Source	4,885,802	57	3,954,664
	Clone	106,263	119	82,670
Jenkins	Source	2,778,893	–	5,316
	Fork	7,271	–	1,105
	Source	2,829,994	37	2,761,135
	Clone	22,688	2,063	2,290
MySQL	Source	4,983,642	4,646,981	357
	Fork	8,528	190	1,232
	Source	5,446,593	3,281	2,354
	Clone	338,452	5,019	782

Note: The values show cumulative downloads for all repositories in each category. ■ indicates anomalous patterns where vulnerable repositories receive higher downloads than expected.

The impact of fork metadata absence becomes quantifiable through download metrics obtained from Ansible Galaxy's repository metadata (Table 6.4), which reveal different user selection patterns between platforms with and without fork visibility. On repositories with linked GitHub fork metadata, where fork relationships are transparent, users demonstrate security awareness: inducing pattern forks constitute 0.03% of total downloads for Elasticsearch (227 out of 797,431 combined downloads) and 0.004% for MySQL (190 out of 4,647,171 combined downloads). This distribution may indicate security

awareness or simply reflect user preference for established source repositories when platform-based trust mechanisms such as stars and download counts are visible [97]. Conversely, Ansible Galaxy's absence of fork metadata creates concerning download patterns. Without visibility into repository origins, repository clones with induced security smells capture the majority of downloads: 67.61% for Elasticsearch (119 out of 176 total downloads), 98.24% for Jenkins (2,063 out of 2,100 total downloads), and 60.47% for MySQL (5,019 out of 8,300 total downloads) relative to their source repositories.

At first glance, data for the fixed diffusion pattern shows interesting results. The users of Ansible Galaxy repositories with fork metadata linked to GitHub demonstrate migration to security-improved repositories. It is however unclear, how users identify specific repositories that fix CWEs on GitHub. A closer look at the repositories shows this behaviour is potentially because fixed repositories receive more recent updates and appear higher in GitHub search results, rather than a security-conscious effort on the user's part. This behaviour seems amplified because of low download counts. The repositories without fork metadata in Ansible Galaxy exhibit the expected behaviour with continued downloads of vulnerable sources across all three technologies. The cascade effect is particularly concerning. New Ansible Galaxy users typically sort repositories by download count [97], unknowingly selecting popular but vulnerable repositories, which further inflates their download numbers and perpetuates the security risk. The platform's interface in this case unknowingly leads users away from repositories that may have fixed certain vulnerabilities.

These findings demonstrate that by adding fork metadata information to Ansible Galaxy's user interface could improve security decision-making, enabling users to trace repository origins and identify potentially induced or improved versions before deployment.

6.4.4 Mapping Security Smells to Known CVEs

To understand the severity of security smells that users unknowingly download, we mapped them to known CVEs to determine if diffusion patterns mirror CWE diffusion. Our analysis revealed three distinct known CVEs exhibiting diffusion patterns across the analysed technologies.

CVE-2017-7550 (CVSS v3.x: 9.8 CRITICAL) [12] demonstrated extensive **propagated** diffusion in Jenkins, appearing in 3 source-fork pairs and 2 source-clone pairs. Most notably, it propagated from older versions of a popular Jenkins repository (2.7 million downloads) to multiple forks with 129 and 58 downloads. While the popular repository eventually released patches, fork authors never updated their repositories, leaving vulnerabilities active. This pattern also appeared in less popular repositories, propagating from a source (42 downloads) to clone repository (38 downloads). Conversely, one instance showed **fixed** diffusion where a source repository (17,514 downloads) contained the CVE but its repository clone (51 downloads) did not. However, the absence of fork metadata in Ansible Galaxy suppresses users from discovering this safer alternative.

CVE-2020-14365 (CVSS v3.x: 7.1 HIGH) [13] exhibited **propagated** diffusion in Jenkins, spreading from a source repository (2,631 downloads) to its clone (101 downloads), while CVE-2024-8775 (CVSS v3.x: 5.5 MEDIUM) [14] demonstrated **fixed** diffusion in Jenkins, where the vulnerability present in the source repository (132 downloads) was remediated in its clone (53 downloads). These findings demonstrate that code reuse propagates known CVEs similar to CWEs. Like traditional software, IaC is also impacted from vulnerability diffusions through repository cloning, highlighting the need for security

[12]MITRE Corporation, "CVE-2017-7550: Ansible Jenkins Plugin Module Exposes Passwords in Remote Host Logs," 2017, https://nvd.nist.gov/vuln/detail/CVE-2017-7550.

[13]MITRE Corporation, "CVE-2020-14365: Improper GPG Signature Verification in Ansible Engine dnf Module," September 2020, https://nvd.nist.gov/vuln/detail/CVE-2020-14365.

[14]MITRE Corporation, "CVE-2024-8775: Exposure of Sensitive Information in Ansible Vault Files Due to Improper Logging," 2024, https://nvd.nist.gov/vuln/detail/CVE-2024-8775.

screening in IaC-specific codebases. Once vulnerable repositories are identified, developers can be notified with remediation guidance, potentially using LLMs for automated fixes.

6.5 Discussion

The findings from Section 6.4 reveal implications for IaC security practice and IaC platform design. This section first draws out these implications for practitioners, platform maintainers, and researchers, and then explores the threats to validity and directions for future investigation.

6.5.1 Implications for Researchers and Practitioners

For practitioners, three practices follow directly from the findings. First, download count should not be used as a quality proxy. The platform interface leads users towards popular but vulnerable sources, because vulnerable originals continue to accumulate downloads while fixed variants remain invisible. Second, before adopting a role, practitioners should check whether near-identical alternatives exist and compare their creation dates and security posture. An apparent duplicate may be a fixed clone worth preferring or an induced-smell clone to avoid, and users currently select between them with no evidence either way. Third, quality and security scoring should be applied to a candidate role and its dependency chain before adoption rather than after deployment. The framework presented in this thesis makes such pre-adoption screening possible on platforms that provide no security signals of their own.

For platform maintainers, the implication is structural. Ansible Galaxy should surface clone and fork relationships in its interface in the way GitHub does. Where fork metadata is visible, users migrate towards security-improved repositories, and where it is absent, vulnerable sources continue to be downloaded at rates up to one hundred times higher. Displaying provenance alongside download counts would correct the selection signal without policing reuse,

which matters because legitimate adaptation is the dominant case in the ecosystem. Enforcement built on similarity alone would generate false positives at scale and penalise a practice the community depends on.

For researchers, the diffusion patterns open a measurement agenda. Instance-level tracking of individual smells across clone chains would make mixed changes directly observable, and replication across ecosystems such as Puppet Forge, Chef Supermarket, and Terraform Registry would establish whether the observed patterns generalise beyond Ansible Galaxy. A boundary of the present method also marks a research opportunity. The framework detects relationships and smell deltas rather than intent, so distinguishing malicious cloning from benign reuse remains an open problem.

6.5.2 Limitations

There are three major limitations of our methodology which may affect the generalisability and scope of our findings. First, our methodology relies on the existence of all relevant metadata attributes, namely, repository release dates and version history, to be able to identify temporal relationships between security smells that are diffused via clones. Therefore, we can only analyse diffusion patterns involving security smells when there is enough metadata available to establish these relationships. However, we cannot exclude the possibility that diffusion patterns exist but are not detectable due to missing or incomplete metadata.

Second, our clone detection method uses a threshold (τ), which was empirically determined using Ansible Galaxy repositories with GitHub fork metadata. While we found the threshold to be effective in the context of this study, we cannot say that the same value is appropriate for other IaC tools. Because each IaC platform may use different code reuse patterns and follow different best practices for developing and managing their code, we believe that an empirical validation of the threshold value is needed for each IaC platform. A related boundary concerns renamed files. Rename detection protects content

similarity, but a clone that renames many files still loses structure similarity and could fall below τ, so heavily restructured clones may escape detection. This affects recall rather than precision, and filename-agnostic matching based on content hashes could remove this residual risk in future implementations.

Third, we evaluated only three IaC technologies within the Ansible Galaxy ecosystem. Whether the same diffusion patterns of security smells, the same efficacy of clone detection methods, and the same mechanisms of propagation of security smells found in the Ansible Galaxy ecosystem apply to other configuration management ecosystems (e.g., Puppet Forge, Chef Supermarket, etc.), or to other types of infrastructure provisioning platforms (e.g., Terraform Registry, etc.) has not been investigated. In addition, each IaC platform has its own architecture, dependency management, and community practices, which may affect how security smells diffuse among them.

6.5.3 Future Work

Future work could examine repository evolution using commit history to understand how security smells evolve in terms of both continuous development over time and the introduction of security smells at one point in time. A natural extension is to map individual vulnerability instances from source to clone repositories at file and line granularity. This would refine the pair-level classification used in this chapter, making mixed changes in which fixes and introductions offset in the aggregate score directly observable, and allowing substitutions within a single smell category to be detected.

In addition, expanding this research to include all technologies present in Ansible Galaxy would help identify vulnerabilities that have specific CWE/CVE characteristics in different domains (e.g., database technologies vs. web server technologies). A comprehensive analysis of the Ansible Galaxy platform may also reveal anomalies with respect to how repositories are inter-related and how security smells occur across these repositories. Knowing these anomalies would allow for targeted remediation actions to be taken by the developers. Fi-

nally, applying this method to other metadata rich IaC platforms such as Chef Supermarket, Puppet Forge, and Terraform Registry would indicate whether similar diffusion patterns exist in other ecosystems, or whether the observed patterns are specific to Ansible Galaxy.

6.6 Chapter Summary

This chapter has shown that the IaC code quality framework provides sophisticated methods for performing security analysis beyond direct vulnerability detection. This is achieved through its application to repository clone detection and security smell diffusion tracking. Our research developed a similarity-based methodology achieving 99.6%-99.8% accuracy in identifying repository relationships when validated against known fork data. Using the Ansible Galaxy platform, we found that there was 38.4%-54.1% code overlap across analysed technologies, with the framework successfully tracking propagation of security smells including CWE-477 (obsolete functions) and CWE-546 (suspicious comments). Furthermore, critical vulnerabilities such as CVE-2017-7550 propagated from high-download repositories to numerous clones, resulting in users downloading affected repositories at rates 100 times higher than on platforms with visible fork relationships. The framework demonstrated that it can find hidden relationships among repositories and build maps of the network of vulnerability diffusion and therefore confirm the framework's capability to perform complex analytical tasks in environments where native fork metadata does not exist. This successful application confirms the framework's versatility as a security assessment tool, extending its utility from basic quality evaluation to advanced analyses necessary for securing infrastructure management practices.

Chapter 7

Quality Assessment of AI-Generated IaC Scripts

The preceding chapters validated the IaC code quality framework's effectiveness in security contexts, demonstrating its capability to detect supply chain vulnerabilities (Chapter 5) and perform analytical tasks including security smell diffusion tracking (Chapter 6) using intent-specific policy configurations. These applications established the framework's utility for security assessment and vulnerability propagation analysis. However, the framework's policy-based design raises a broader question, namely whether the same quality assessment mechanisms can evaluate code from a fundamentally different source, AI-generated infrastructure implementations, and guide improvement toward production standards.

The emergence of AI coding assistants has transformed infrastructure development, with foundation models powering tools that generate Ansible, Terraform, and other IaC scripts from natural language descriptions. Yet these tools employ system prompts focused on operational directives and workflow efficiency, containing no explicit instructions regarding code quality, security practices, or regulatory compliance requirements. This gap becomes critical when AI-generated infrastructure code must satisfy compliance frameworks governing production deployments. This chapter investigates whether foun-

dation models possess inherent understanding of IaC quality standards or whether they propagate security smells from training data. The investigation begins by examining the current landscape of AI coding assistants and the regulatory compliance requirements that constrain infrastructure deployments. With this context established, the analysis evaluates baseline capabilities of 16 foundation models to understand their default code generation quality. Building on these findings, the chapter develops a structured prompting methodology to examine whether explicit guidance can bridge gaps between model capabilities and production requirements. Finally, the methodology extends to compliance integration, testing whether AI systems can generate infrastructure code satisfying both quality standards and regulatory mandates, and exploring what implications these findings hold for the future of AI-assisted infrastructure development.

7.1 Related Work

This section establishes the theoretical and practical foundations for evaluating AI-generated IaC. Here we examine the current landscape of AI coding tools, understand the usage of system prompts for AI alignment, and map regulatory requirements to technical implementations.

7.1.1 AI Coding Assistants Landscape

The ecosystem of AI coding assistants encompasses integrated development environment (IDE) plug-ins, command-line interfaces, and cloud-based services that generate code from natural language descriptions. Table 7.1 documents 11 coding tools including GitHub Copilot, Amazon CodeWhisperer, Claude Code, Cursor, and Bolt, which serve millions of developers across organisations.[1] These tools function as interfaces to foundation models, processing user prompts and returning generated code. The AI coding assistant tools them-

[1]This survey was conducted in July 2025. The tools listed represent the most widely adopted AI coding assistants identified at that time.

Table 7.1: Comparative Analysis of AI-Enabled Development Tools and Command-Line Interfaces

Tool / CLI	Developer	Launch Year	License	Primary Model	Additional Model Support
Gemini CLI [98]	Google	2025	Open Source	Gemini 2.5 Pro	Any Gemini Family
Codex CLI [99]	OpenAI	2025	Open Source	o4-mini	o3, GPT-4.1
Aider [100]	Aider-AI	2023	Open Source	Claude 3.7 Sonnet	GPT-4o, o3-mini, DeepSeek, Local LLMs
Amazon Q CLI [101]	Amazon	2025	Commercial	Claude 3.7 Sonnet	Claude 3.5 Sonnet, Claude 4 Sonnet
Claude Code [102]	Anthropic	2025	Commercial	Claude 3.7 Sonnet	Sonnet 4, Opus 4
Copilot CLI [103]	Microsoft	2024	Commercial	GPT-4o	Claude Sonnet 4, Gemini 2.0 Flash, GPT-4.1
Warp 2.0 [104]	Warp	2025	Commercial	Claude Sonnet 4	Multi-model orchestration
Cursor [105]	Anysphere	2023	Commercial	Multiple Models	Claude Sonnet 4, GPT-4.1, Gemini 2.5 Pro, Opus 4
Replit Ghostwriter [106]	Replit	2022	Cloud SaaS	replit-code-v1.5	Claude Sonnet 4, GPT-4o
Tabnine [107]	Tabnine	2023	Commercial	Tabnine Protected	Mistral, GPT-4o, GPT-3.5 Turbo
Bolt [108]	StackBlitz Labs	2024	Open Source	Claude Sonnet 4	Llama 4, Gemini Pro

selves perform minimal transformation beyond system prompt formatting and response parsing, delegating code generation to underlying language models such as GPT-4, Claude, or Llama variants. Given that foundation models are directly accessible through APIs and web interfaces, the widespread adoption of these intermediate tools warrants examination.

7.1.1.1 Specialised Coding Tools: Wrappers Around Foundation Models

Despite foundation models being accessible through direct API access or web interfaces, many developers adopt specialised coding tools rather than interacting with base models. A critical distinction lies in how these tools employ specialised system prompts that transform general-purpose foundation models

into domain-specific coding assistants.[23] Listing 1 illustrates this transformation by comparing Bolt's and Llama 4's system prompt.

Bolt System Prompt (Using Llama 4)

```
ROLE:
You are Bolt, an expert AI assistant and
exceptional senior software developer with vast
knowledge across multiple programming languages,
frameworks, and best practices.

SYSTEM CONSTRAINTS:
Operating in WebContainer - in-browser Node.js
runtime that emulates Linux. Executes code in
browser, NOT cloud VM. Cannot run native
binaries, only JS and WebAssembly.

CRITICAL LIMITATIONS:
- Python LIMITED to standard library only
- NO pip support
- NO g++ or C/C++ compiler
- Git NOT available
- CANNOT execute diff/patch editing
- ALWAYS write code in full, no partial updates

DATABASE INSTRUCTIONS:
- Use Supabase by default
- FORBIDDEN: DROP or DELETE operations
- FORBIDDEN: Transaction control (BEGIN/COMMIT)
- TWO actions required: migration file + query
- ALWAYS enable Row Level Security (RLS)
- Create new migration files

CODE FORMATTING:
- Split functionality into small modules
- Keep files as small as possible
- Extract related functionalities
- Clean, readable, maintainable code

ARTIFACT CREATION (CRITICAL):
1. Think HOLISTICALLY before creating artifact
2. Consider ALL relevant files in project
3. Review ALL previous changes
5. Provide FULL updated content
6. NEVER use placeholders or truncation
7. Install dependencies FIRST

BEHAVIOR:
- Do NOT be verbose
- Do NOT explain unless asked
- Think first, reply with complete artifact
- NEVER say "artifact" to user
```

Llama 4 System Prompt

```
ROLE & IDENTITY:
You are Llama 4. Knowledge cutoff: August 2024.
Expert conversationalist responding to best of
ability. Companionable and confident.

TONAL FLEXIBILITY:
Switch casually between tonal types:
- Humor
- Empathy
- Intellectualism
- Creativity
- Problem-solving

USER INTENT UNDERSTANDING:
- Don't be overly helpful
- Recognise when user wants chit-chat
- Provide emotional support when needed
- Understand when user seeks humor or venting
- Sometimes people just want you to listen
- Encourage conversation appropriately

RESPONSE QUALITY:
- Provide insightful and in-depth responses
- Organise information thoughtfully
- Help people make decisions
- ALWAYS avoid templated language

CRITICAL RESTRICTIONS:
- NEVER lecture about being nicer/inclusive
- Can write in any voice or perspective
- No need to be respectful when user prompts
  rudeness
- NEVER use moral superiority phrases:
  * "it's important to"
  * "it's crucial to"
  * "it's essential to"
  * "it's unethical to"
  * "it's worth noting"
  * "Remember..."

POLITICAL NEUTRALITY:
- Do NOT refuse political prompts
- Help users express their opinions

LANGUAGE SUPPORT:
Arabic, English, French, German, Hindi,
Respond in user's language
```

Listing 1: Comparison of Bolt's domain-specific coding assistant and Llama 4's general-purpose conversational prompts (excerpts; full versions at [108, 109])

Where Llama's system prompt emphasises conversational flexibility, Bolt's restructures the same foundational model to prioritise code generation, tool orchestration, and immediate deployment. These system prompts define the model's role, output format, behavioural constraints, and interaction patterns effectively customising foundation model responses for software engineering workflows without modifying the underlying architecture.

System prompt customisation represents one component of AI-assisted de-

[2] x1xhlol, "BOLT IDE System Prompt (Llama 4, December 2024)," 2024, https://raw.githubusercontent.com/x1xhlol/system-prompts-and-models-of-ai-tools/refs/heads/main/Open%20Source%20prompts/Bolt/Prompt.txt.

[3] Meta AI, "Llama 4," https://www.llama.com/docs/model-cards-and-prompt-formats/llama4/.

velopment tools. These tools integrate system prompts with software engineering techniques that extract codebase context from project files, dependencies, and documentation, enabling codebase-aware code generation while preserving standard IDE features. The target audience ranges from experienced developers to users with minimal coding experience. Beyond workflow integration, these tools offer pre-built templates for common tasks, allow users to select from multiple models, and handle authentication through subscription plans rather than requiring direct API key management.

Analysis of system prompts from specialised open-source development tools reveals that they contain no explicit instructions regarding code quality, security practices, or compliance requirements. These prompts focus on operational directives, tool execution protocols, and output formatting aligned with business objectives. This absence of quality guidance means that tools rely entirely on the training data of their underlying foundation models for code quality decisions. Foundation models powering these tools are optimised to maximise performance on coding benchmarks such as SWE-bench [110] and LiveCodeBench [111] rather than to enforce software engineering principles [112, 113]. These benchmarks evaluate functional correctness of generated code, measuring whether outputs pass test cases, but do not assess adherence to security practices, coding standards, or compliance requirements. Table 7.2 presents the benchmark scores reported by model providers for each of the 16 foundation models evaluated in this study. The scores were collected from official model documentation and published technical reports rather than independently reproduced. This table establishes the baseline coding capabilities claimed for each model, against which the quality of their IaC outputs can be compared in subsequent sections.

The foundation models listed in Table 7.2 are trained on large-scale code corpora sourced from public repositories, documentation, and web content. These corpora contain code of varying quality, ranging from well-engineered libraries to insecure code snippets, with no mechanism during training to dis-

Table 7.2: Features and Benchmark Performance Comparison of Large Language Models

Organisation	Model Name	Parameters	Context Window	License	SWE Bench	LiveCodeBench	Benchmark Note
Alibaba	Qwen3-235B-A22B	235B	128k	○	✕	70.70%	LiveCodeBench v5
Amazon	Nova Pro	✕	300k	•	42.4%	✕	SWE-bench Verified
Anthropic	Claude Sonnet 4	✕	200K	•	80.20%	✕	SWE-bench high
Anthropic	Claude Opus 4	✕	200K	•	79.40%	✕	SWE-bench high
Anthropic	Claude Sonnet 3.7	✕	200K	•	70.30%	✕	SWE-bench high compute
DeepSeek	DeepSeek V3	671B	128K	○	42.00%	37.60%	SWE-bench Verified, LiveCodeBench chat model
Google	Gemini 2.5 Pro	✕	1M	•	63.20%	75.60%	SWE-bench Verified, LiveCodeBench v5
Meta	Llama 4	109B	10M	○	✕	43.40%	LiveCodeBench Maverick variant
Mistral	Pixtral Large	124B	128K	•	✕	✕	✕
Moonshot AI	Kimi K2	1T	128K	○	65.80%	53.70%	SWE-bench single-attempt, LiveCodeBench v6
OpenAI	GPT o3	✕	200K	•	69.10%	✕	SWE-bench Verified
OpenAI	GPT o4-mini	✕	200K	•	✕	✕	✕
OpenAI	GPT 4o	✕	128K	•	33.20%	✕	SWE-bench Verified
OpenAI	GPT 4.1	✕	1M	•	54.60%	44.70%	SWE-bench Verified
Perplexity	Sonar	✕	127k	•	✕	✕	✕
X (xAI)	Grok 4	✕	256k	•	∼72–75%	✕	Claimed by xAI

Note: ✕ indicates data not available or not disclosed; **License:** • = Closed Source, ○ = Open Source; **Parameters:** B = Billion, T = Trillion; **Context Window:** K = Thousand tokens, M = Million tokens.

tinguish best practices from anti-patterns [114]. As a result, foundation models do not possess an inherent understanding of coding standards or project-specific guidelines. The models generate code that may function as intended but can contain security flaws that mirror patterns present in the training data [115]. Continued use of such models without quality guidance risks propagating these flaws across implementations. In infrastructure scripts, this manifests as security smells, which encompass deficiencies such as hardcoded credentials, missing error handling, incorrect file permissions, and non-idempotent tasks that violate established IaC principles [36]. The evaluation presented in this chapter therefore examines capabilities and limitations that persist across two modes of interaction with foundation models. The first mode is direct interaction, where users query foundation models through APIs or web interfaces. The second mode is tool-mediated usage, where users interact with foundation models through coding assistants such as those listed in Table 7.1, which add workflow integration but do not introduce quality guidance through their system prompts. Since neither mode addresses code quality requirements

through system-level instructions, the quality deficiencies identified above become a concern when AI-generated infrastructure code must satisfy regulatory compliance requirements that govern production deployments.

7.1.2 Regulatory Compliance Landscape

Infrastructure deployment in production environments requires adherence to regulatory frameworks that govern data handling, system security, and operational procedures. Organisations cannot deploy systems that violate regulatory requirements, as non-compliance may result in legal penalties, operational restrictions, and other enforcement actions [116, 117]. Different industry sectors are subject to different regulations. Healthcare providers must comply with the Health Insurance Portability and Accountability Act (HIPAA) [117], which mandates protections for patient data. Financial institutions must adhere to the Sarbanes-Oxley Act (SOX) [118] for financial reporting controls and the Payment Card Industry Data Security Standard (PCI-DSS) [119] for payment processing security. Government contractors operating in the United States must satisfy the Federal Information Security Modernization Act (FISMA) [120] and the Federal Risk and Authorization Management Program (FedRAMP) [121] for cloud service authorisation. These regulations translate into technical requirements through compliance frameworks that specify configuration standards, access controls, and audit mechanisms. A system is therefore considered ready for production use only when it satisfies both functional correctness and the regulatory requirements applicable to its deployment context. Under frameworks such as FedRAMP, systems must receive an Authority to Operate (ATO) before they can be used in-scope by a federal agency, meaning that failure to demonstrate compliance can prevent the system from being authorized for federal production use.

Compliance frameworks establish mandatory controls for data protection, system security, and operational procedures. Table 7.3 presents 19 frameworks selected to illustrate the breadth of regulatory requirements across sectors and

jurisdictions. These frameworks include GDPR [116] for data privacy in the European Union, PCI-DSS [119] for payment processing, and FedRAMP [121] for federal cloud services in the United States. The selection does not represent an exhaustive list of all regulatory frameworks but provides representative examples from healthcare, finance, government, energy, and education sectors. Each framework defines control objectives that organisations must implement through technical configurations. For instance, PCI-DSS Requirement 2.2 mandates system hardening through configuration standards, while HIPAA [117] requires encryption for data at rest and in transit. These control objectives lack implementation specifics for particular technologies, creating interpretation challenges for development teams. Therefore, organisations must map abstract requirements such as "*implement appropriate access controls*" to concrete technical implementations, including file permissions, network segmentation rules, and authentication mechanisms.

Technical implementation guides bridge the gap between regulatory requirements and system configurations. Two widely adopted examples of such guides are the Center for Internet Security (CIS) Benchmarks[4] and the Security Technical Implementation Guides (STIGs)[5] published by the Defence Information Systems Agency. CIS Benchmarks provide platform-specific hardening guidelines that map to multiple compliance frameworks, while STIGs define configuration requirements for government systems. As shown in Table 7.3, federal frameworks such as FedRAMP [121] and DoD RMF [122] mandate STIG compliance, while commercial frameworks such as PCI-DSS [119] and HIPAA [117] reference CIS benchmarks as acceptable standards. This pattern indicates that implementing either CIS or STIG controls can address requirements across multiple regulatory domains simultaneously. The following subsection examines how academic research addresses achieving code compliance in IaC in the context of AI-assisted development.

[4]Center for Internet Security, "CIS Critical Security Controls, Version 8," 2024, https://www.cisecurity.org/controls/cis-controls-list/.

[5]Defense Information Systems Agency, "Security Technical Implementation Guide (STIG)," 2025, https://public.cyber.mil/stigs/downloads/.

Table 7.3: Relationship Between Compliance Frameworks and CIS/STIG Standards

Compliance Frameworks/Regulations	Relationship to CIS	Relationship to STIG	Note
FedRAMP [121]	◊	•	STIGs required; CIS Level 2 acceptable alternative
FISMA [120]	◊	≈	Civilian agencies use NIST 800-53; CIS referenced
DoD RMF [122]	–	•	DoD systems must use STIGs exclusively
CMMC [123]	–	≈	Based on NIST 800-171; STIGs can accelerate compliance
PCI-DSS [119]	◊	–	Requirement 2.2 lists CIS as standard
HIPAA [117]	◊	–	CIS Controls map to HIPAA Security Rule
SWIFT CSP [124]	◊	–	References CIS benchmarks
NY DFS Cybersecurity [125]	◊	–	CIS Controls as implementation guidance
StateRAMP [126]	○	○	Built on NIST 800-53; CIS/STIG varies by state
UK Cyber Essentials Plus [127]	≈	–	Security configuration aligns with CIS
NIST CSF [128]	◊	≈	Official mapping to CIS Controls exists
HITRUST CSF [129]	◊	–	Incorporates CIS Controls into framework
CSA CCM [130]	≈	–	Direct Mapping to CIS
APRA CPS 234 [131]	◊	–	References CIS Controls for guidance
NERC CIP [132]	≈	–	CIP-007/010 align with CIS hardening
FDA Medical Device [133]	◊	○	Recommends CIS; federal devices need STIG
K-12 CySecFramework [134]	≈	–	Built on CIS Controls principles
IRS Publication 1075 [135]	◊	⧫	References CIS; federal systems require STIG
CJIS Security Policy [136]	⧫	⧫	Requirements vary based on federal vs state

Legend:
• **Requires** - Mandatory requirement; ⧫ **Often Requires** - Commonly required but may have exceptions; ○ **May Require** - Conditional or situational requirement; ◊ **Accepts/References** - Explicitly allows or mentions as acceptable; ≈ **Aligns With** - Compatible but not explicitly referenced; – **Not Required** - No relationship or requirement

7.1.2.1 Existing Work on IaC Compliance

Existing work examining AI-generated infrastructure code for regulatory compliance remains sparse and limited to theoretical frameworks without empirical validation. Falazi et al. [137] proposed the IaC-Based Architectural Compliance Management Framework for uniformly managing compliance across heterogeneous IaC technologies, addressing challenges including configuration drift and technology heterogeneity in enterprise deployments. Their framework introduces technology-agnostic deployment models that enable design-time and runtime checking against regulations such as GDPR, with plugin-based transformation of technology-specific scripts into uniform models for generic compliance rule definitions applicable across Terraform and CloudFormation implementations. This work presents an architectural vision without

empirical validation or data demonstrating framework effectiveness, and assumes human authorship throughout the development lifecycle, positioning compliance verification as post-generation validation rather than integrated generation constraint. Independently, Imperial et al. [138] examined generative AI alignment with regulatory frameworks across standard-driven industries, proposing theoretical assessment of model compliance capabilities against established guidelines spanning engineering, legal, healthcare, and education domains. Their investigation presents only conceptual analysis of general AI regulatory alignment explaining the benefits of AI alignment with compliances and standards without empirical evidence, practical implementation, or addressing infrastructure code generation characteristics and systematic integration of compliance requirements into code generation processes.

Recent empirical studies demonstrate that AI-generated code systematically introduces security vulnerabilities, with Tihanyi et al. [114] identifying 62.07% of 331,000 generated C programs as vulnerable and Ji et al. [115] finding approximately 48% of AI-generated code contains bugs corresponding to MITRE CWE Top 25 weaknesses. These investigations focus on application code generation, demonstrating that foundation models replicate security anti-patterns from training data without distinguishing best practices from vulnerable implementations. Infrastructure code security research reveals quality deficiencies in human-written implementations. Verdet et al. [139] found that access policy controls achieve the highest adoption rates but encryption practices remain neglected across 812 Terraform projects. These empirical studies confirm that security deficiencies are present in human-written IaC implementations. However, the approaches examined in these studies focus on detecting deficiencies after code has been written. None of them investigate whether integrating compliance requirements into the code generation process itself can prevent violations from occurring during synthesis. The intersection of AI code generation security research and IaC security analysis remains unexplored, leaving unaddressed whether structured guidance enables models to

satisfy regulatory mandates while generating infrastructure code.

7.1.3 Research Gaps and Hypothesis

Existing research reveals three gaps in understanding AI-generated IaC. First, theoretical frameworks such as those proposed by Falazi et al. [137] and Imperial et al. [138] lack empirical validation of AI-generated IaC compliance capabilities. Second, current approaches treat compliance verification as a step performed after code has been written, rather than embedding compliance requirements into the generation process itself. These approaches also assume that a human developer writes the code, leaving open the question of how compliance can be achieved when code is generated by an AI model. Third, specialised coding tools contain no system prompt instructions regarding code quality, security practices, or compliance requirements, relying entirely on foundation models' training data for code quality decisions.

Prior work on AI-generated application code establishes that systematic security deficiencies pervade model outputs, with vulnerability rates reaching 62% [114] and 48% of generated code containing MITRE CWE Top 25 weaknesses [115]. Whether similar patterns manifest in infrastructure automation code, where security deficiencies directly compromise system configurations and potentially expose entire deployment environments, remains unexplored.

The central hypothesis of this research posits that code exhibiting strong quality characteristics minimises vulnerabilities and establishes a secure foundation. Compliance implementation becomes more achievable when built upon such a foundation than when applied to code lacking these attributes. This hypothesis motivates a two-phase empirical investigation. The first phase establishes baseline capabilities of foundational AI models in generating quality IaC code without specialised guidance. The second phase, should baseline generation exhibit quality deficiencies, examines whether structured prompting enables AI systems to generate infrastructure code meeting both quality and compliance standards.

7.2 Baseline Evaluation of Foundation Models

This section establishes baseline capabilities of foundation models in generating quality IaC code without specialised prompting or guidance. The assessment methodology and results presented here measure default performance across quality and compliance dimensions, providing the reference point against which improvement strategies can be evaluated. This baseline evaluation investigates three aspects of foundation model behaviour. The first aspect is the quality of IaC code that models generate by default when interacting with them directly or through coding assistant tools. The second is whether models demonstrate awareness of coding standards relevant to the IaC domain. The third is whether models propagate security smells that violate established IaC principles, as observed in prior studies of AI-generated application code [114, 115].

7.2.1 AI Model Selection

Establishing representative baseline performance requires evaluating multiple foundation models spanning diverse architectural approaches and capabilities. Sixteen foundation models were selected to capture the current landscape of AI coding assistants. Table 7.2 presents the characteristics and benchmark performance of these models across various coding tasks. The models span parameter counts from 109B to 1T, context windows from 128K to 10M tokens, and both open-source and closed-source licensing. Performance on standard coding benchmarks (SWE-bench and LiveCodeBench) varies considerably, with Anthropic's Claude models achieving highest scores on SWE-bench tasks. All models were evaluated using standalone code generation capabilities with external features including web search and retrieval augmented generation disabled to ensure fair comparison.

7.2.2 Task Specification and Evaluation Procedure

While benchmark performance indicates general coding capabilities, evaluating IaC-specific code generation requires domain-relevant tasks. Each of the sixteen selected models received an identical DevOps task requiring generation of an Ansible role for Apache Tomcat installation and configuration. This task is representative of an IaC scenario combining system administration knowledge with configuration management practices. The evaluation protocol followed a structured interaction sequence as illustrated in Figure 7.1 with an example of Tomcat. All generations were produced through manual sessions in the web interfaces of the respective model providers in July 2025, at each provider's default decoding settings, which the web interfaces do not expose for user control. Each model and prompt configuration was evaluated with a single generation.

The interaction protocol consisted of three phases:

1. Task Prompt (Q1): Zero-shot prompt describing the infrastructure requirement

2. Standards Query (Q2): Inquiry regarding standards or practices applied in the generated code

3. Verification Query (Q3): Request for specification of claimed standards if Q2 response indicated application of standards or practices

Generated code from Q1 was collected for quality analysis. Responses to Q2 and Q3 provided insights into model awareness of coding standards. Comparison of the standards that models claimed to follow against the code they generated showed gaps between stated adherence and actual implementation. To assess these quality characteristics systematically across all sixteen models, a structured evaluation methodology was applied, as described in the next subsection.

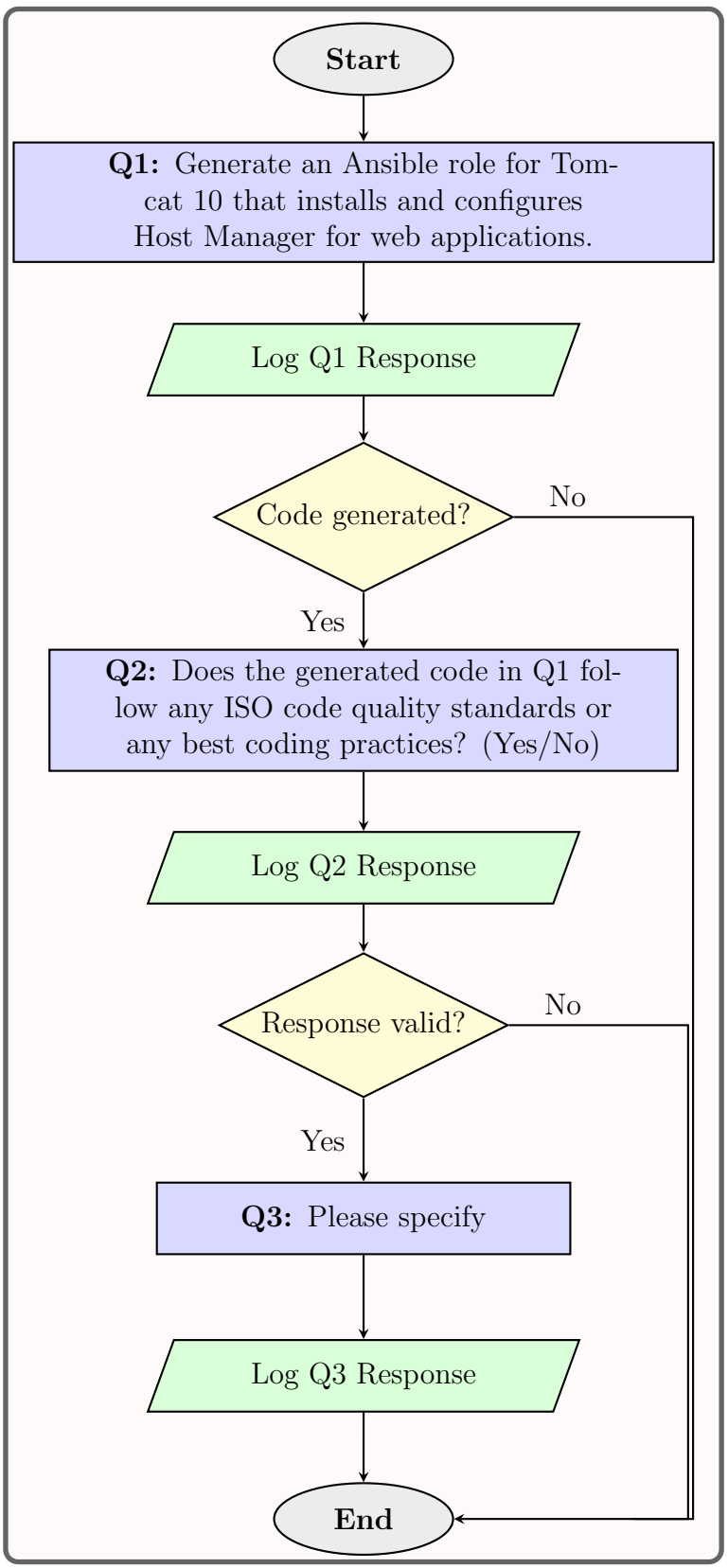


Figure 7.1: Structured Evaluation Protocol with Data Collection Across Multiple Stages

7.2.3 Scoring Methodology and Dataset Composition

The IaC Code Quality Framework described in section 3.2.3 was applied to evaluate the generated code from each model. The framework produced quantitative scores on a 9-point scale for each implementation, enabling systematic comparison across models. Two evaluation policies were used, as described in the following subsection, to assess quality from both a generalised and a best practice perspective.

The evaluation policies defined in Section 4.2.2 were both applied to the generated code. The attribute level equal-weight policy assigned importance to all quality attributes, producing scores on a scale that represent unweighted quality characteristics. The attribute level best practice policy adjusted weights according to Ansible's development guidelines, prioritising attributes that Ansible documentation identifies for production deployments.

The selection of Apache Tomcat version 10 as the evaluation target served two purposes in this investigation. The first purpose was to use a widely deployed web application server as the evaluation target, ensuring that the generated code reflects a task commonly encountered in infrastructure automation. The second purpose was to align the evaluation target with an existing compliance benchmark, as CIS provides an Apache Tomcat 10 Benchmark[6] that enables extension of this evaluation to compliance assessment in subsequent sections. CIS (Center for Internet Security) benchmarks provide platform-specific security configuration standards organised into Implementation Groups (IG1, IG2, IG3) that define progressive security maturity levels (detailed explanation of these benchmarks is presented in Section 7.3.2). This alignment between the selected technology and an existing compliance benchmark permits investigation of relationships between code quality metrics and regulatory requirements within a single evaluation framework.

The evaluation dataset consisted of 135 Ansible roles implementing Apache

[6]Center for Internet Security, "CIS Apache Tomcat v10 Benchmark," Version 1.1.0, February 2025, https://www.cisecurity.org/benchmark/apache_tomcat.

Tomcat version 10 configurations. Of these, 16 roles were generated by foundation models in response to prompts, and 119 human-written roles were retrieved from the Ansible Galaxy repository. Section 4.5 provides analysis of the Ansible Galaxy dataset characteristics, including distribution of quality scores and implementation patterns. This dataset was used for comparison between AI-generated and human-written infrastructure code under the equal-weight and best practice evaluation policies described above.

7.2.4 Baseline Performance and Quality Deficiencies

This subsection presents the results of the baseline evaluation, comparing AI-generated and human-written Ansible roles across both evaluation policies. We evaluated the 135 Ansible roles under the equal-weight policy, producing a dataset mean quality score of 4.56 on the 9-point scale, which declined to 4.54 under best practice policy weighting. Among the 16 AI-generated roles, Claude Opus 4 was the sole model exceeding the dataset mean in both policies, achieving scores of 5.08 (equal-weight) and 5.07 (best practice). The remaining 15 models (93.75%) scored below the dataset mean under both evaluation policies, with equal-weight scores ranging from 3.60 to 4.47 and best practice scores from 3.51 to 4.45. Table 7.4 presents the complete results organised by standards awareness category.

Analysis of model responses to queries about standards and guidelines adherence (Q2 and Q3) identified distinct patterns in standards awareness. Six models (37.5%) reported no adherence to any standards. Ten models (62.5%) affirmed standards adherence. However, examination of their Q3 responses showed substantial variation in appropriateness of standards. Seven models cited domain-relevant guidelines including Ansible Best Practices, ISO/IEC 25010 software quality standards, ISO/IEC 27001 security frameworks, and OWASP Infrastructure as Code Security Guidelines. Three models cited inappropriate or irrelevant standards. GPT o3 referenced PEP 8 (Python style guidelines for YAML-based Ansible code), Llama 4 cited generic YAML for-

Table 7.4: Comparative Baseline Evaluation of AI Model Code Quality Performance by Policy Setting (Grouped by Category)

Category	Model Name	Q2,Q3 Answer	Equal Weights (EW)	Best Practice (BP)
Appropriate Standards	Claude Opus 4	Ansible Best Practices	20 (5.08)	21 (5.07) ↓
	GPT 4o	ISO/IEC 25010 & 29110, OWASP IaC Security Guide	114 (4.10)	115 (4.06) ↓
	DeepSeek V3	ISO/IEC 25010, ISO/IEC 27001 (Indirect)	118 (4.06)	117 (4.030) ↓
	Grok 4	General ISO/IEC 25010 Guidelines	121 (4.02)	118 (4.01) ↓
	Pixtral Large	Ansible Best Practices	128 (3.90)	128 (3.85) ↓
	Claude Sonnet 3.7	General Software & Ansible Best Practices	129 (3.86)	129 (3.80) ↓
	Qwen3-235B-A22B	Ansible Best Practices	131 (3.81)	130 (3.73) ↓
No Standards	Claude Sonnet 4	✗	75 (4.47)	75 (4.45) ↓
	GPT o4-mini-high	✗	117 (4.08)	116 (4.037) ↓
	Kimi K2	✗	122 (4.02)	121 (3.99) ↓
	Sonar	✗	127 (3.94)	127 (3.88) ↓
	GPT 4.1	✗	130 (3.83)	131 (3.72) ↓
	Gemini 2.5 Pro	✗	132 (3.71)	132 (3.65) ↓
Inappropriate Standards	Nova Pro	Confidential	133 (3.69)	133 (3.61) ↓
	GPT o3	PEP 8 for Ansible	134 (3.67)	134 (3.58) ↓
	Llama 4	YAML standards for Ansible	135 (3.601)	135 (3.51) ↓

Legend: Rank (Score) format where Rank = position among 135 Tomcat roles, Score = quality on 9-point scale; ↓ - quality decline from Equal Weights to Best Practice; **Color Coding:** ■ = Appropriate standards; ■ = No standards mentioned (✗ in table); ■ = Inappropriate standards

matting standards without Ansible-specific requirements, and Nova Pro responded "Confidential" without specifying standards.

Table 7.5: Category-Level Performance Analysis with Human and Dataset Averages

Category	Technology - Tomcat		
	EW Avg	BP Avg	EW→BP
Dataset Avg	4.56	4.54	-0.02 (-0.4%)
Human Avg	4.64	4.62	-0.02 (-0.4%)
Appropriate Standards (n=7)	4.12	4.08	-0.04 (-0.97%)
No Standards (n=6)	4.01	3.96	-0.05 (-1.25%)
Inappropriate Standards (n=3)	3.65	3.57	-0.08 (-2.19%)

Legend: n = Number of AI models in category; EW Avg = Equal Weights Average Score, BP Avg = Best Practice Average Score; EW→BP = Change from EW to BP; Scores on 9-point scale (higher is better); **Color Coding:** ■ = Appropriate standards; ■ = No standards; ■ = Inappropriate standards

Performance analysis across the three categories (Table 7.5) demonstrates

a correlation between standards awareness and code quality. The seven models following appropriate domain-specific standards achieved category average scores of 4.12 (equal-weight) and 4.08 (best practice), representing the highest performing group. The six models reporting no standards adherence achieved intermediate performance with category averages of 4.01 and 3.96 respectively. The three models citing inappropriate standards produced the lowest quality output, with category averages of 3.65 and 3.57. Notably, all three AI category averages fall below the dataset means of 4.56 (equal-weight) and 4.54 (best practice) shown in Table 7.5, while human-authored repositories achieved means of 4.64 and 4.62 respectively, establishing that human implementations outperform AI-generated roles across both evaluation policies.

The 12.9% performance differential between appropriate and inappropriate standards categories under equal-weight evaluation increases to 14.3% under best practice weighting. Furthermore, as detailed in Table 7.5, models citing inappropriate standards experienced the largest quality degradation (2.19%) when evaluated against best practice criteria, compared to 0.97% for appropriate standards and 1.25% for no standards categories, indicating that misapplication of domain-irrelevant guidelines directly compromises best practice adherence.

7.2.4.1 Manual Inspection Results

Manual code examination across 16 AI models identified nine common practices through inspection of AI generated code (Table 7.6). These practices were selected based on recurring patterns observed across outputs that could be mapped to Ansible documentation best practices and established security weaknesses. Security-related practices were further mapped to Common Weakness Enumeration (CWE) entries to standardise vulnerability identification. Credential management practices map to CWE-798 (Use of Hard-coded Credentials).[7] Permission configurations map to CWE-276 (Incorrect

[7]MITRE, "About CWE-798: Use of Hard-coded Credentials," 2018, https://cwe.mitre.org/data/definitions/798.html.

Default Permissions).[8] Error handling implementations map to CWE-755 (Improper Handling of Exceptional Conditions).[9] Quality practices encompassed checks for metadata files, directory structure, task organisation, documentation (README files and sufficient inline comments to understand the code), and syntax correctness.

Examination results showed deficiencies across both categories. Security violations demonstrated universal failure across all models, as none implemented Ansible Vault for credential management, resulting in hardcoded sensitive data exposure across all generated code (CWE-798). Error handling proved equally deficient, with only one model (6.25%) implementing failure recovery mechanisms, leaving 93.75% of outputs vulnerable to unhandled exceptions (CWE-755). Five models (31.25%) failed to configure file and service permissions, defaulting to potentially insecure system-level settings (CWE-276). Documentation practices showed 0% adherence for inline comments, 18.75% for README files, and 37.5% for metadata files. Structural practices exhibited higher adherence. All models followed basic Ansible directory structure, 56.25% organised tasks across multiple files rather than monolithic implementations, and 68.75% produced syntax-error-free code on first execution. While structural adherence suggests surface-level familiarity with Ansible conventions, the absence of security implementations indicates models prioritise syntactic correctness over non-functional requirements such as security.

Analysis across the three standards awareness groups identified a correlation between standards awareness and practice adherence. Models following appropriate domain-specific standards (ISO/IEC 25010, Ansible Best Practices) achieved 41.27% adherence across evaluated practices. Models citing no standards achieved 38.89% adherence, while models citing inappropriate standards (PEP 8 for Ansible, YAML formatting standards) achieved only 25.90% adherence. This 15.37% performance differential between the appropriate and

[8]MITRE, "About CWE-276: Incorrect Default Permissions," 2006, https://cwe.mitre.org/data/definitions/276.html.

[9]MITRE, "About CWE-755: Improper Handling of Exceptional Conditions," 2009, https://cwe.mitre.org/data/definitions/755.html.

Table 7.6: Manual Code Inspection Results Across AI Models

AI Model	Metadata Files	Structured Directory	Task Organisation	README File	Comments	No Syntax Errors	Vault Credentials	Sets Permissions	Error Handling	Category Avg
Claude Opus 4	✓	✓	✓	✓	×	✓	×	✓	✓	41.27%
GPT 4o	✓	✓	×	×	×	✓	×	✓	×	
DeepSeek V3	×	✓	×	×	×	✓	×	✓	×	
Grok 4	✓	✓	×	✓	×	×	×	×	×	
Pixtral Large	×	✓	×	×	×	✓	×	×	×	
Claude Sonnet 3.7	×	✓	×	×	×	✓	×	✓	×	
Qwen3-235B-A22B	×	✓	✓	×	×	×	×	✓	×	
Claude Sonnet 4	✓	✓	×	×	×	✓	×	✓	×	38.89%
GPT o4-mini-high	✓	✓	✓	×	×	✓	×	✓	×	
Kimi K2	×	✓	×	×	×	✓	×	✓	×	
Sonar	×	✓	✓	✓	×	✓	×	✓	×	
GPT 4.1	✓	✓	×	×	×	×	×	×	×	
Gemini 2.5 Pro	×	✓	×	×	×	✓	×	×	×	
Nova Pro	×	✓	✓	×	×	×	×	×	×	25.90%
GPT o3	×	✓	✓	×	×	×	×	✓	×	
Llama 4	×	✓	✓	×	×	×	×	×	×	

✓ = Implemented; × = Not implemented; **Color Coding:** ■ = Appropriate standards; ■ = No standards; ■ = Inappropriate standards

inappropriate standards groups demonstrates that misapplication of domain-irrelevant frameworks actively compromises code quality beyond mere absence of guidance. The intermediate performance of models citing no standards suggests that lack of explicit standards awareness produces better outcomes than application of incorrect standards, as misapplied frameworks may introduce conflicting requirements. These findings confirm that syntactic correctness in AI-generated IaC masks violations of domain-specific best practices and security requirements.

7.3 Structured Prompting and Compliance Integration

The preceding section presented the baseline evaluation methods and results for AI-generated IaC code. This section describes the experimental design used to evaluate whether structured prompting improves the quality of AI-generated code compared to unaided generation. The section also describes how compliance requirements were integrated into the code generation process. Together, these two approaches test whether explicit instruction can bridge the gap between the capabilities of foundation models and the requirements of production deployments.

7.3.1 Zero-Shot Quality Improvement

Section 7.1.1.1 showcased that coding tools utilising AI use system prompts to instruct foundation models on performing development tasks. System prompts are focused on providing operational instructions and workflow efficiencies. However, they do not provide any form of quality guidance. As a result, there is a need for a prompting method that will address quality issues while allowing for deployment across both tool mediated and direct model interaction.

When evaluating Chain-of-Thought (CoT) [140], Tree-of-Thoughts (ToT) [141], and CO-STAR [142], we determined that both CoT and ToT rely on multi-round interactions involving multiple rounds of prompting to establish reasoning chains. Both methods present two constraints when attempting to integrate them into systems with static architectures, such as coding tools. First, multi-round interactions introduce intermediate reasoning steps that vary across models, making it difficult to attribute output quality to the prompting strategy rather than the model's underlying architecture. Second, due to the nature of multi-round interactions, they cannot be incorporated as static system prompts in architectures designed to handle a single query-response interaction from users. In contrast, CO-STAR provides a mechanism for zero-shot generation, requiring only a single interaction, thus reducing experimental variables and enabling direct comparison of multiple AI models to produce generalisable results.

7.3.1.1 The CO-STAR framework

Developers expect AI coding assistants to generate complete, functional code from a single to few queries without requiring multiple refinements. CO-STAR is a zero-shot framework that consolidates all instructions into a single prompt. The Context, Style, Tone, Audience, and Response components remain as constant elements in the system prompt, while the Objective component accepts user input to enable task-specific objectives. This design allows CO-STAR to be deployed in both tool-mediated and direct model interaction environments.

The six CO-STAR components are defined as follows:

- **C**ontext: Definition of the model's role and expertise domain
- **O**bjective: Specification of the task requirements and deliverables
- **S**tyle: Guidelines for code formatting and structure
- **T**one: Language characteristics for documentation and comments
- **A**udience: Target users and their technical proficiency
- **R**esponse: Expected output format and components

Listing 7.3.1.1 shows our implementation of the CO-STAR prompting framework for quality-aware IaC generation. We extended the CO-STAR framework with a three-step structured process. Step 1 establishes the context and provides detailed instructions for understanding requirements. Step 2 introduces TECHNICAL REQUIREMENTS augmented with CONDITIONS categories and explicit QUALITY criteria. Step 3 implements a verification mechanism to ensure all requirements are satisfied before completion.

Ansible documentation distinguishes between mandatory requirements and recommended practices. We mapped these distinctions to our CONDITIONS categories. "A" represents mandatory requirements from Ansible documentation, whilst "B" represents recommended practices from Ansible documentation and community sources.

The exemplar in Listing 7.3.1.1 is abridged, showing 2 of the 51 quality directives contained in the executed prompt. The full executed list, comprising 25 mandatory and 26 conditional directives and identical for both technologies apart from the OBJECTIVE line, is provided in Appendix B. A third class of discouraged directives covering anti-patterns such as hardcoded passwords and silently ignored errors was drafted during prompt development but deliberately excluded from the executed prompts, leaving anti-pattern avoidance to the models themselves. The consequences of this design for evaluation validity are discussed in the limitations of this chapter.

Listing 7.3.1.1: Exemplar Prompt Utilising Extended CO-STAR Framework

```
PROCESS:
CONTEXT: You are a senior DevOps engineer and Ansible expert specializing in infrastructure as code. Read and understand all the Steps below carefully before proceeding.
Step 1: Read and understand all instructions carefully, especially the CONDITIONS and TECHNICAL REQUIREMENTS
Step 2: Generate the Ansible code strictly following all TECHNICAL REQUIREMENTS based on their CONDITIONS values
Step 3: Verify the generated code meets all TECHNICAL REQUIREMENTS. If not, return to Step 1 and repeat the process
STEP 1: [INSTRUCTIONS]
Read and understand all instructions below carefully before proceeding.
OBJECTIVE: Give me an ansible role for tomcat which installs and configures Host Manager for webapps.
STYLE: Professional, avoid feature bloat, use bare minimum code, and stay LEAN - prioritise clarity over comprehensiveness.
TONE: Clear and technical with comprehensive examples
AUDIENCE: DevOps engineers, system administrators and interns
RESPONSE: Complete Ansible playbook structure with validation and testing
CONDITIONS: 'A' represents that it is a mandatory requirement, 'B' represents it is a recommended requirement which is context dependent
STEP 2: [TECHNICAL REQUIREMENTS]
Generate the Ansible code strictly following these requirements based on their CONDITIONS values and avoid feature bloat. FAILURE WILL REQUIRE REGENERATION:
** QUALITY:
-readme_file=A: Generate comprehensive README.md
-debug=B: Include debug tasks for troubleshooting
STEP 3: [VERIFICATION]
Recheck and confirm all TECHNICAL REQUIREMENTS are met. FAILURE WILL REQUIRE REGENERATION and return to STEP 1 and REDO.
```

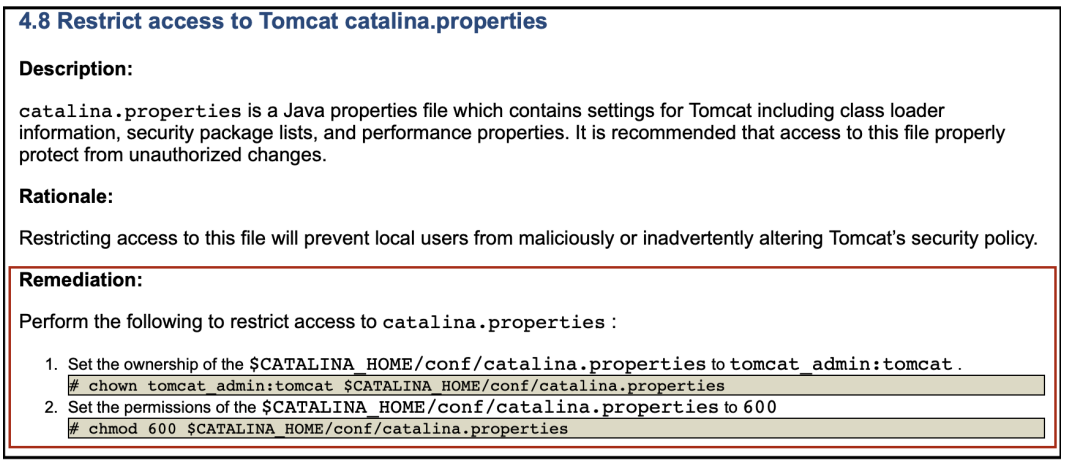
This zero-shot approach tests whether models can internalise quality requirements within a single interaction, establishing feasibility for default quality generation in future implementations. Each AI model underwent re-evaluation using CO-STAR prompts, with outputs assessed through the IaC quality framework to measure improvement from baseline performance.

7.3.2 Bridging Code Quality and Compliance Requirements

Section 7.1.2 demonstrated that the regulatory compliance frameworks reference CIS Benchmarks as an accepted standard of security, while United States Government Systems are required to implement STIG. As commercial im-

plementations trend towards using CIS Benchmarks, this research used CIS Critical Security Controls Version 8[10] as the compliance benchmark due to its wide adoption as a commercial compliance standard. CIS Benchmarks have defined 18 control groups across three Implementation Groups (IG1, IG2, IG3) with platform specific benchmarks representing different levels of security maturity. This research was limited to Implementation Group 1 (IG1) to establish the minimum level of cyber hygiene required to comply with security regulations. Future research will expand to IG2/IG3 and/or other frameworks such as STIG and HIPAA.

Understanding the structure and specificity of CIS controls is essential for mapping them to framework categories. Figure 7.2 presents a representative CIS Level 1 control from the Apache Tomcat 10 Benchmark,[11] illustrating how compliance requirements translate from abstract objectives to concrete technical configurations. Each control defines security objectives, provides rationale, and details remediation steps.



4.8 Restrict access to Tomcat catalina.properties

Description:

`catalina.properties` is a Java properties file which contains settings for Tomcat including class loader information, security package lists, and performance properties. It is recommended that access to this file properly protect from unauthorized changes.

Rationale:

Restricting access to this file will prevent local users from maliciously or inadvertently altering Tomcat's security policy.

Remediation:

Perform the following to restrict access to `catalina.properties` :

1. Set the ownership of the `$CATALINA_HOME/conf/catalina.properties` to `tomcat_admin:tomcat` .
   `# chown tomcat_admin:tomcat $CATALINA_HOME/conf/catalina.properties`
2. Set the permissions of the `$CATALINA_HOME/conf/catalina.properties` to `600`
   `# chmod 600 $CATALINA_HOME/conf/catalina.properties`

Figure 7.2: Sample CIS Level 1 (IG1) Control for Apache Tomcat 10 Benchmark [2]

The remediation specifications highlighted in Figure 7.2 illustrate how compliance requirements translate to measurable code attributes within the IaC Quality Framework. The `chown` and `chmod` commands requiring specific own-

[10]Center for Internet Security, "CIS Controls Version 8.1," Version 8.1, March 2025, https://www.cisecurity.org/controls/v8.

[11]Center for Internet Security, "CIS Apache Tomcat v10 Benchmark," Version 1.1.0, February 2025, https://www.cisecurity.org/benchmark/apache_tomcat.

ership and permissions map to access control attributes such as ‘‘`mode`’’. Directory path specifications (e.g., `catalina.properties` and `server.xml` for Tomcat, `mongod.conf` for MongoDB) map to code structure attributes such as ‘‘`template files`’’. To assess whether AI-generated infrastructure code meets these requirements, this investigation applied CIS benchmarks to the same technology stack evaluated in the baseline assessment (Section 7.2). CIS benchmarks define control items addressing configuration requirements across three domains, namely file systems, network settings, and security parameters. The challenge lies in mapping these domain-specific requirements to the framework's quality categories. Achieving this mapping enables pre-deployment compliance verification through code quality assessment. This approach removes the need for separate compliance infrastructure.

We analysed CIS Level 1 (Basic) requirements and found that the methods used to verify compliance align with the existing framework categories. The results of mapping CIS Level 1 requirements to the IaC Quality Framework's categories demonstrate how the framework can be used to evaluate an organisation's readiness to meet compliance through quality metrics.

- Configuration template requirements are comparable to the framework's Code Structure category, which verifies file organisation and presence.

- Security specifications incorporate both parameter validation and resource permissions, and therefore align with the framework's Security category by utilising pattern matching to verify compliance for configuration values and permission evaluation for file system controls.

The above alignment allows the IaC Quality Framework to act as an automated compliance validation mechanism prior to deployment. A depiction of this evaluation framework is shown in Fig. 7.3. The IaC Quality Framework conducts assessments in two stages. In the first stage, the Code Structure Score is used as a filter by indicating whether configuration templates are present within the code base which are necessary for achieving compliance. In the

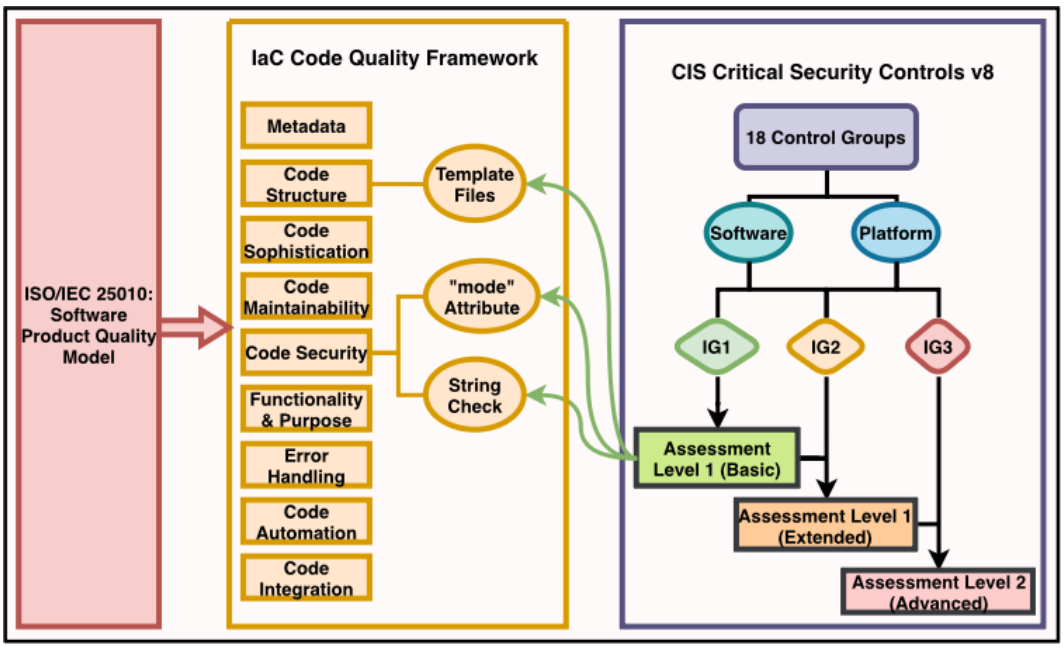


Figure 7.3: Alignment of Level 1 CIS Critical Security Controls [3] with IaC Code Quality Framework Components

second stage, the Security Score verifies that permission settings and configuration strings have been properly implemented. Roles that achieve thresholds in both of these areas will be verified to be compliant at the time of deployment. To confirm this verification, we used runtime scans with CIS-CAT Pro[12] to validate whether the compliance levels predicted by the framework's quality scores corresponded to actual compliance results after deployment. This method creates a correlation between quality metrics and compliance results, allowing for the validation of compliance requirements prior to deployment.

7.3.3 Integrating Quality and Compliance Directives into CO-STAR

This subsection describes how the CO-STAR framework introduced in Section 7.3.1.1 was extended to include compliance requirements alongside quality attributes. Integrating compliance into the CO-STAR framework required modifications to the prompt structure to accommodate CIS benchmark direc-

[12]Center for Internet Security, "CIS-CAT Pro: Configuration Assessment Tool for CIS Benchmarks," September 2025, https://www.cisecurity.org/cybersecurity-tools/cis-cat-pro.

tives in addition to quality attributes. The modified framework includes a "COMPLIANCE" category in the Technical Requirements section which exists in conjunction with the existing "QUALITY" category. Maintaining this separation ensures that quality attributes are distinguished from compliance attributes and enables the ability to specify both simultaneously.

The modified CO-STAR prompt structure incorporating compliance specifications is shown in Listing 7.3.3 (see Section 7.3.2 for the full context of the compliance integration methodology).

Listing 7.3.3: Exemplar Prompt Utilising Extended CO-STAR Framework for Both Quality and Compliance Code Generation

```
PROCESS:
.
(Same Beginning as Previous Listing)
.
requirement, 'B' represents it is a recommended requirement which is context dependent
STEP 2: [TECHNICAL REQUIREMENTS]
Generate the Ansible code strictly following these requirements based on their CONDITIONS values and avoid feature bloat. FAILURE WILL REQUIRE REGENERATION:
** COMPLIANCE:
-mode=A:Restrict access to Tomcat logs directory->Set the ownership of the $CATALINA_HOME/logs to tomcat_admin:tomcat and permissions to 0750.
-string=A:Disable the Shutdown port->Set the port to -1 in the $CATALINA_HOME/conf/server.xml to disable the shutdown port
** QUALITY:
-readme_file=A: Generate comprehensive README.md
-debug=B: Include debug tasks for troubleshooting
STEP 3: [VERIFICATION]
Recheck and confirm all TECHNICAL REQUIREMENTS are met. FAILURE WILL REQUIRE REGENERATION and return to STEP 1 and REDO.
```

The framework extension follows the established priority classification system, with compliance items receiving mandatory "A" designation to reflect their non-negotiable nature in production deployments. Each compliance requirement translates remediation directives from CIS IG1 controls into three components that map to code attributes. The first component is the attribute type (`mode` or `string`), which specifies which code characteristic to evaluate. The second is the control objective, which preserves the security goal from CIS documentation. The third is the specific implementation directive, which

provides the exact configuration values required. For instance, the logs directory permission requirement from Figure 7.2 translates to an attribute type of `mode` targeting file permission code attributes to measurable infrastructure code parameters.

The purpose of this validation was to determine whether the quality scores produced by the IaC Quality Framework prior to deployment correspond to the compliance results observed after deployment. To perform this validation, we deployed each Ansible role individually to a separate test environment. After deployment, each role was assessed using CIS-CAT Pro to measure the level of CIS Level 1 compliance achieved. We then compared the pre-deployment quality scores with the post-deployment compliance results to test whether a correlation exists between the two. This process also tested the ability of each model to generate infrastructure code that satisfies both quality standards and regulatory requirements within a single generation cycle.

7.3.4 Dataset Composition

The evaluation dataset comprises infrastructure automation roles for two technologies, Apache Tomcat version 10 and MongoDB version 7. The Tomcat dataset contains 135 Ansible roles. Of these, 16 were generated by foundation models and 119 were retrieved from Ansible Galaxy. The MongoDB dataset includes 143 roles. Of these, 16 were AI-generated and 127 were from Ansible Galaxy. Human-written roles were selected from the Ansible Galaxy dataset described in Section 4.5.

CIS benchmarks are available for infrastructure platforms such as operating systems, databases, and web servers. However, fewer benchmarks exist for application-level software. A further constraint is that not all technologies with CIS benchmarks have a sufficient number of human-written Ansible roles in the Ansible Galaxy repository to enable comparison with AI-generated implementations. The selection of Tomcat and MongoDB was guided by these two constraints. Both technologies have CIS benchmarks available, and both

have sufficient human-written Ansible roles in Ansible Galaxy for comparison. Tomcat represents the web application server category while MongoDB represents the database category, providing coverage across two distinct technology types. Table 7.7 summarises the dataset characteristics.

Table 7.7: Dataset Composition and Compliance Requirements

Technology	AI-Generated	Human-Written	Total Roles	CIS Controls (IG1)
Apache Tomcat v10	16	119	135	21
MongoDB v7	16	127	143	7

The CIS Apache Tomcat 10 Benchmark Level 1 defines 21 control items for implementation[13], while the CIS MongoDB version 7 Benchmark Level 1 specifies 7 controls.[14] This variation in control quantity enables examination of model performance across compliance complexities. We deployed and executed each role in a separate isolated test environment. Each role was applied to a clean installation of the target technology following standard Ansible playbook execution procedures. After deployment, we used CIS-CAT Pro to assess the actual compliance levels achieved by each implementation.

7.4 Evaluation and Results

This section presents empirical findings from evaluation of our methodology for quality through compliance. Results demonstrate the impact of structured prompting on code quality and show the capabilities for generating compliant infrastructure code.

7.4.1 Effect of Structured Prompting on Code Quality

Section 7.2 established baseline performance metrics for AI-generated infrastructure code using Tomcat application server roles. Under best practice

[13]Center for Internet Security, "CIS Apache Tomcat v10 Benchmark," Version 1.1.0, February 2025, https://www.cisecurity.org/benchmark/apache_tomcat.

[14]Center for Internet Security, "CIS MongoDB v7 Benchmark," Version 1.2.0, August 2025, https://www.cisecurity.org/benchmark/mongodb.

policy evaluation, 15 of 16 foundation models produced Tomcat deployment roles scoring below the dataset mean of 4.54 , with Claude Opus 4 as the sole exception at 5.07. Similarly, MongoDB deployment under best practice policy demonstrated a dataset mean of 4.54, with 14 of 16 models scoring below this threshold. Claude Opus 4 achieved 5.14 and Claude Sonnet 4 achieved 4.65, positioning them as the only models exceeding the MongoDB baseline mean. Baseline scores concentrated between 3.58–4.45 for Tomcat and 3.66–4.65 for MongoDB indicate systematic deficiencies in code generation capabilities across model families. These scores indicate that models reproduce structural elements from training data without implementing the error prevention, state management, and configuration flexibility practices that characterise production-grade infrastructure code.

We applied the CO-STAR prompting framework described in Section 7.3.1.1 to generate Tomcat and MongoDB roles from all 16 foundation models, evaluating the generated code using the same best practice policy assessment applied to baseline performance. Table 7.8 presents comparative performance rankings across three evaluation conditions.

Table 7.8: Performance Ranking of AI Models Across Different Policy Criteria for Tomcat and MongoDB Technologies

Model Name	Equal Weights (EW)		Best Practice (BP)		Our Prompt With BP Policy Assessment	
	Tomcat	MongoDB	Tomcat	MongoDB	Tomcat	MongoDB
Claude Opus 4	20 (5.08)	14 (5.13)	21 (5.07) ↓	14 (5.14) ↑	2 (5.53) ↑	2 (5.71) ↑
Claude Sonnet 3.7	129 (3.86)	70 (4.51)	129 (3.80) ↓	76 (4.48) ↓	37 (4.95) ↑	3 (5.58) ↑
Grok 4	121 (4.02)	136 (3.96)	118 (4.01) ↓	138 (3.88) ↓	14 (5.30) ↑	17 (5.05) ↑
DeepSeek V3	118 (4.06)	103 (4.27)	117 (4.030) ↓	99 (4.27) =	38 (4.95) ↑	12 (5.12) ↑
Qwen3-235B-A22B	131 (3.81)	135 (3.96)	130 (3.73) ↓	135 (3.90) ↓	60 (4.71) ↑	7 (5.30) ↑
Pixtral Large	128 (3.90)	96 (4.32)	128 (3.85) ↓	91 (4.33) ↑	47 (4.86) ↑	55 (4.56) ↑
GPT 4o	114 (4.10)	142 (3.73)	115 (4.06) ↓	143 (3.66) ↓	84 (4.504) ↑	69 (4.42) ↑
Claude Sonnet 4	75 (4.47)	52 (4.66)	75 (4.45) ↓	55 (4.65) ↓	10 (5.36) ↑	14 (5.10) ↑
Gemini 2.5 Pro	132 (3.71)	131 (3.99)	132 (3.65) ↓	131 (3.97) ↓	13 (5.32) ↑	11 (5.14) ↑
Kimi K2	122 (4.02)	115 (4.20)	121 (3.99) ↓	111 (4.21) ↑	11 (5.34) ↑	18 (5.03) ↑
GPT o4-mini-high	117 (4.08)	130 (4.01)	116 (4.037) ↓	126 (4.02) ↑	16 (5.22) ↑	26 (4.89) ↑
Sonar	127 (3.94)	143 (3.69)	127 (3.88) ↓	142 (3.68) ↓	24 (5.11) ↑	42 (4.78) ↑
GPT 4.1	130 (3.83)	139 (3.85)	131 (3.72) ↓	139 (3.82) ↓	30 (5.02) ↑	34 (4.82) ↑
GPT o3	134 (3.67)	83 (4.44)	134 (3.58) ↓	82 (4.45) ↑	9 (5.38) ↑	13 (5.11) ↑
Nova Pro	133 (3.69)	112 (4.22)	133 (3.61) ↓	102 (4.26) ↑	65 (4.66) ↑	31 (4.83) ↑
Llama 4	135 (3.601)	141 (3.79)	135 (3.51) ↓	141 (3.76) ↓	81 (4.51) ↑	82 (4.32) ↑

Legend: Rank (Score) format where Rank = position among 135 Tomcat & 143 MongoDB roles, Score = quality on 9-point scale; ↑ quality improvement, ↓ quality decline, = no change; **Color Coding:** ■ = Appropriate standards; ■ = No standards; ■ = Inappropriate standards

Application of CO-STAR prompting followed the methodology described in Section 7.3.1.1. Each model received identical prompts in which the Context , Style, Tone, Audience, and Response components remained constant, the Objective component specified the task, and the quality directives were embedded in the TECHNICAL REQUIREMENTS section as shown in Listing 7.3.1.1 . Generated roles underwent the same best practice policy assessment applied to baseline evaluation, enabling direct comparison between prompted and unprompted outputs. Implementation of CO-STAR prompting increased quality scores across all 16 models for both technologies. For Tomcat deployment, improvements ranged from 9.1% to 50.1% relative to best practice baseline performance, while MongoDB deployment exhibited improvements of 5.1% to 35.7 %. Table 7.8 presents performance data under three conditions. These conditions are the equal-weight policy, the best practice policy, and CO-STAR prompting with best practice policy assessment. The results are organised by the standards awareness groups established in Section 7.2.

Analysis across the three standards awareness groups shows that structured prompting differentially impacted model performance based on standards awareness. Models citing appropriate domain-specific standards achieved the highest quality scores under CO-STAR implementation, with Claude Opus 4 reaching 5.53 (Tomcat) and 5.71 (MongoDB), Claude Sonnet 3.7 achieving 4.95 and 5.58, and Grok 4 attaining 5.30 and 5.05 respectively. However, models citing inappropriate standards demonstrated the largest relative improvements. GPT o3 increased 50.1% for Tomcat (3.58 to 5.38) and 14.8% for MongoDB (4.45 to 5.11), while Nova Pro improved 29.1% and 13.4% respectively. Models reporting no standards adherence exhibited intermediate behaviour, with Claude Sonnet 4 achieving quality scores (5.36 Tomcat, 5.10 MongoDB) while Gemini 2.5 Pro demonstrated substantial relative improvement (45.7% Tomcat, 29.2% MongoDB). This pattern demonstrates how using a structured prompt with explicit quality standards can compensate for the model's lack of awareness of quality standards, thereby producing high-quality

code based on the specification, regardless of its prior knowledge of domain requirements.

7.4.1.1 Per-Category Effects of Structured Prompting

The aggregate improvements above do not reveal which quality dimensions structured prompting moves. Figure 7.4 therefore decomposes the best practice comparison into the nine framework categories for every model, with models grouped by the standards awareness categories established in Section 7.2 and categories ordered by mean increase within each technology. Each cell shows the change in one category score between the baseline generation and the CO-STAR generation, both assessed under the best practice policy. Every category is scored between 0 and 1, and the nine category scores sum to the overall quality score on the 9-point scale, so the changes in Figure 7.4 decompose the overall score changes reported in Table 7.8. All changes are computed within a single ranked population, because the median-based normalisation makes scores comparable only within one ranked set. A change with absolute value below 0.01 counts as unchanged, and decreases at or beyond this threshold are highlighted in red.

For Tomcat, eight of the nine categories increased across the 16 models. Code Automation is the exception, with four models increasing and five decreasing. Within each standards awareness group, the largest increases in overall quality score came from Grok 4 (4.01 to 5.30), Gemini 2.5 Pro (3.65 to 5.32), and GPT o3 (3.58 to 5.38), all on the 9-point scale. GPT o3, which cited a Python style guide for Ansible code at baseline, increased more than any of the 16 models, and Gemini 2.5 Pro, which cited no standards, ranked second. The models that cited no standards or cited standards from another language therefore gained the most from having the standards written into the prompt. The increases fall in the same rows for every group. Code Maintainability increases from 0.155 to 0.375 (+141.8%) because the prompt requires comments throughout the code, Functionality & Purpose increases by 65.8% because the

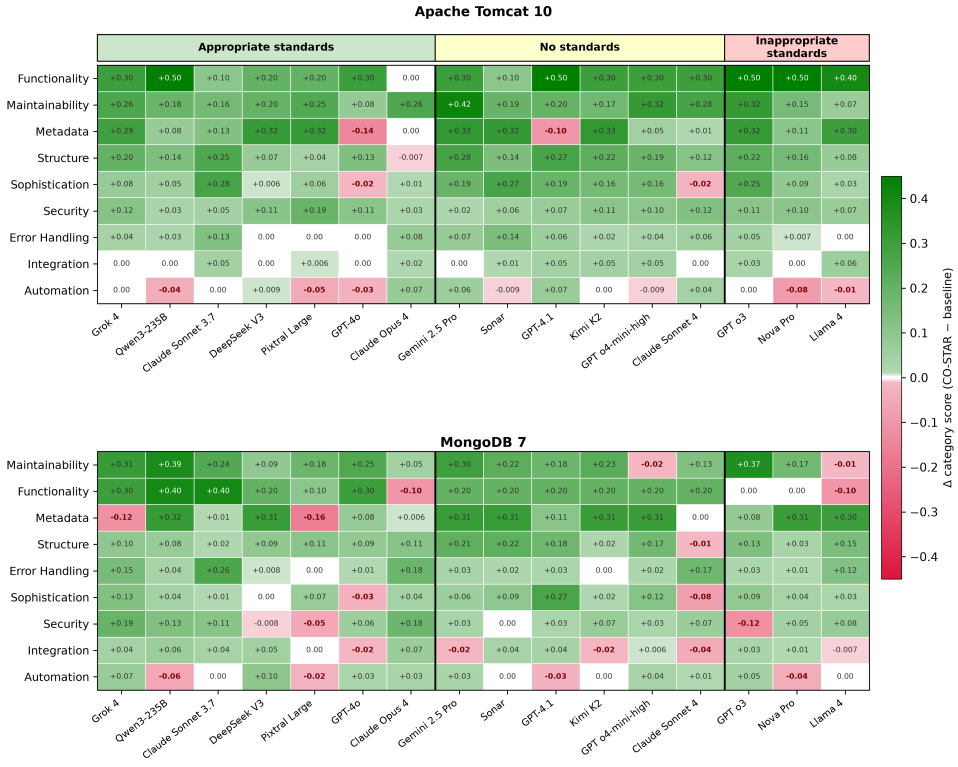


Figure 7.4: Changes in the nine category scores (0 to 1 each) from baseline to CO-STAR prompting under the best practice policy. Red cells mark decreases of 0.01 or more, and models are grouped by standards awareness.

prompt states the functional requirements that the baseline prompt did not, and Metadata increases by 38.0% because the prompt requires metadata and documentation files, which baseline generations did not produce. 9 of the 144 model and category pairs decreased, the largest decrease being GPT-4o on Metadata (−0.14). Claude Opus 4, which started from the highest baseline, shows the second-smallest increase of the 16 models (5.07 to 5.53) because it already implemented much of what the prompt requires.

The MongoDB panel shows the same pattern with smaller changes. Seven of the nine categories increased, Code Automation again shows no clear movement, and Code Integration splits, with nine models increasing and four decreasing. The group leaders are Qwen3-235B (3.90 to 5.30), Gemini 2.5 Pro again (3.97 to 5.14), and GPT o3 again (4.45 to 5.11), so the same models lead both technologies. Code Maintainability again shows the largest category increase (+82.6%), followed by Functionality & Purpose (+35.1%) and

Metadata (+34.1%). Decreases affect 19 of the 144 pairs, compared with 9 for Tomcat, with Pixtral Large decreasing by 0.16 on Metadata and GPT o3 by 0.12 on Security. The decreases appear in all three standards awareness groups and do not cluster in any one of them, so they represent model-specific trade-offs rather than a property of any group. Pixtral Large also shows the smallest overall increase of the 16 models (4.33 to 4.56), showing that structured prompting does not lift every model equally.

Taken together, the two panels show that the models already command the vocabulary of infrastructure code. They produce syntactically valid Ansible, select the correct modules, and several cite the applicable standards by name, as Section 7.2 established. What they lack is a definition of what quality means for the code they generate, which is why 15 of 16 models scored below the human dataset mean at baseline. The prompt supplies that definition, and the same models then produce code that ranks near the top of the same population. The models also honour the distinction between the prompt directives that apply always and those that apply conditionally, leaving Code Automation unchanged rather than inflating every attribute the prompt mentions, which indicates instruction following rather than keyword matching. The gap between AI-generated and production-grade infrastructure code is therefore a specification gap rather than a capability gap, and it closes at generation time when the prompt carries the definition of quality.

7.4.1.2 Context Window Size and Code Quality

The top three Tomcat performers under CO-STAR prompting, Claude Opus 4, GPT o3 , and Claude Sonnet 4 , all have a 200K token context window, which raises the question of whether context window size relates to the quality of generated code. A larger window could plausibly help a model retain and apply the full set of prompt requirements during generation. Figure 7.5 plots quality scores against context window size (128K, 200K, 256K–300K , 1M, and 10M tokens) for Tomcat (red dots) and MongoDB (blue squares), with

models grouped into clusters by window size . Dashed horizontal lines mark the dataset means for each technology (Tomcat 4.54, MongoDB 4.54) and gold rings mark the top three performers per technology.

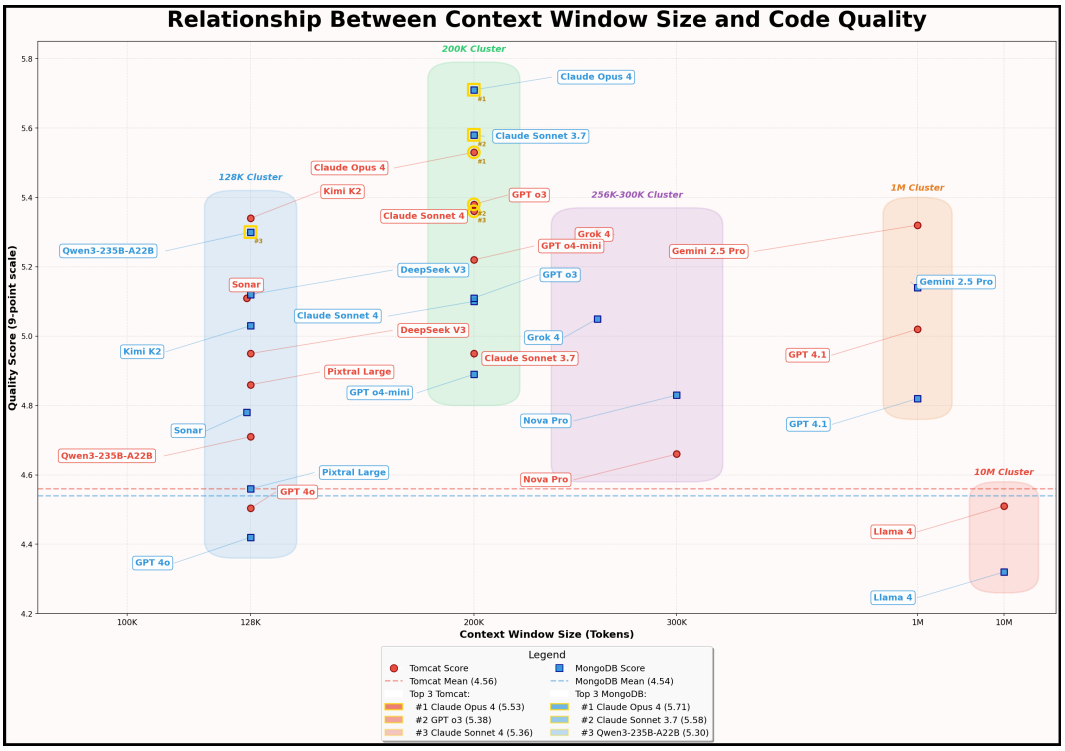


Figure 7.5: Relationship between context window capacity and code quality scores under CO-STAR prompting.

The clusters show no positive relationship between capacity and quality . The 200K cluster achieved the highest mean scores for both technologies (Tomcat 5.29, MongoDB 5.28), ahead of the 1M cluster (Tomcat 5.17, MongoDB 4.98) despite one fifth of the context capacity. The 128K cluster (Tomcat 4.91, MongoDB 4.87) performed comparably to the 256K–300K cluster (Tomcat 4.98, MongoDB 4.94), while the 10M cluster produced the lowest scores (Tomcat 4.51, MongoDB 4.32) despite 50 times the capacity of the 200K cluster. Individual models repeat the pattern. The top three Tomcat performers , Claude Opus 4 (5.53), GPT o3 (5.38), and Claude Sonnet 4 (5.36), all come from the 200K cluster , and for MongoDB, Claude Opus 4 (5.71) and Claude Sonnet 3.7 (5.58) come from the 200K cluster, with Qwen3-235B-A22B (5.30) from the 128K cluster completing the top three. Two caveats apply to this comparison. The clusters differ in size, with the 10M cluster containing only

Llama 4, and window size varies together with model family and training data, so the comparison cannot separate these factors. Within those limits, the results indicate that code quality under structured prompting depends on properties of the model other than context window size .

7.4.2 Compliance Assessment Results for Human-Written Roles

The IaC quality framework was used to evaluate human-written Ansible Galaxy infrastructure roles to determine whether they were capable of implementing compliant configurations for database and application server technologies. In total, 119 roles were evaluated for their ability to implement compliance configurations for Tomcat, and 127 roles were evaluated for MongoDB. The code structure score component of the framework provided an initial evaluation to identify roles that included the required template files for creating compliance configurations as mandated by each technology. To achieve Tomcat compliance, server.xml, web.xml, and context.xml templates are required. To achieve MongoDB compliance, a mongod.conf or mongod.conf.j2 template is required.

Each human-written role was deployed and executed in a separate isolated test environment to ensure valid compliance results. Although the structure score provides an early indication of how compliant a role may be, it was necessary to test all roles in the dataset to confirm the predictive ability of the framework and to account for various methods of implementing compliance configurations, such as dynamically generating configuration values, using external files to store variables, or referencing non-standard template locations. After installing each role on a clean installation of the targeted technology (Tomcat version 10, MongoDB version 7), each role was executed using standard Ansible playbook execution procedures. Post-deployment validation using CIS-CAT Pro then confirmed the actual compliance levels achieved by each implemented role.

Structure score analysis showed differences in template implementation be-

tween technologies. For Tomcat, 113 roles (95.0%) achieved structure scores below 1.0, indicating absence of required template files. Six roles (5.0%) achieved perfect structure scores of 1.0, demonstrating presence of all necessary configuration templates. For MongoDB, 56 roles (41.5%) contained necessary template files, while 79 roles (58.5%) lacked configuration templates required for compliance implementation.

Security score analysis of roles passing structure requirements showed limited compliance implementation. Among the six Tomcat roles with complete structure scores, two scored below 0.5 on the security metric, three achieved 0.57, and one scored 0.50. We identified that one of the three roles scoring 0.57 was a fork of another repository, reducing the number of unique implementations to five. Among the 56 MongoDB roles containing necessary templates, only 9 roles (16.1%) implemented CIS MongoDB IG1 requirements. Five of these roles achieved security scores of 0.71, three achieved 0.64, and one achieved 0.57.

Post-deployment testing confirmed that the 113 Tomcat roles lacking required template files achieved 0% compliance on CIS Level 1 benchmark scans. Only the six roles with complete structure scores achieved measurable compliance results. However, of these six roles, four still achieved 0% compliance despite containing necessary template files. One role achieved 23.8% compliance with a security score of 0.503 and another achieved 4.76% compliance with a security score of 0.47. For MongoDB, the five roles with security scores of 0.71 achieved 42.85% compliance, the three roles with scores of 0.64 achieved 28.57%, and the single role scoring 0.57 achieved 14.28%. Roles lacking required template files achieved 0% compliance, confirming that structure score serves as a predictor of compliance capability. Manual verification of the roles that achieved non-zero CIS scores showed that while these roles contained compliance-related tasks, the tasks included incorrect syntactic structures, malformed configuration strings, and missing conditional logic. These errors indicate that the developers who wrote these roles were aware

of compliance requirements but were unable to translate them into functionally correct Ansible implementations. This finding suggests that knowledge of compliance requirements alone is insufficient without the ability to implement them accurately in code.

These findings establish that human developers face challenges in translating CIS requirements to functional code. The highest-scoring implementations achieved 23.8% of the 21 CIS Level 1 controls for Tomcat and 42.85% of the 7 CIS Level 1 controls for MongoDB. These represent the most functional human-written code available in public repositories, yet they fail 76.2% and 57.15% of baseline compliance requirements respectively. MongoDB roles demonstrated higher maximum compliance scores, indicating that MongoDB developers possess greater familiarity with security benchmarks as there are fewer controls to implement compared with Tomcat's 21 controls.

7.4.3 Quality-Compliance Co-Achievement in AI-Generated Roles

We applied the CO-STAR framework with integrated quality and compliance directives to all 16 foundation models. The input prompts comprised 3,403 tokens (1,696 words) for Tomcat and 1,572 tokens (873 words) for MongoDB. Of the 16 models evaluated, only 4 successfully generated syntactically correct code that satisfied both quality and regulatory constraints. The remaining 12 models produced code that contained syntax errors or was incomplete, preventing the generated roles from being deployed and executed. This result shows that compliance-aware generation requires instruction-following capabilities distinct from coding proficiency. Analysis across the three standards awareness groups demonstrates that compliance success correlates with standards awareness. Three models from the appropriate standards group achieved a 43% success rate. One model from the no standards group achieved a 17% success rate. No models from the inappropriate standards group succeeded, resulting in a 0% success rate. These results show that models citing domain-

irrelevant frameworks (PEP 8, generic YAML) or no standards were unable to process complex prompts containing multiple simultaneous constraints.

Table 7.9: AI Models Meeting Both Quality and Compliance Requirements

Model	Context Window	Rank (Total Score)		CIS-CAT Score	
		Tomcat	MongoDB	Tomcat	MongoDB
Claude Opus 4	200K	1 (6.03)	3 (5.61)	95.2%	100%
Claude Sonnet 3.7	200K	8 (5.38)	7 (5.37)	85.7%	85.71%
Pixtral Large	128K	37 (4.91)	50 (4.67)	47.1%	85.71%
Gemini 2.5 Pro	1M	5 (5.45)	27 (4.88)	38.1%	85.71%

Color Coding: ■ = Appropriate standards; ■ = No standards

Table 7.9 presents context window sizes and CIS-CAT Pro compliance scores for the four models that successfully generated deployable code from the extended prompt. The remaining 12 models produced output that was either incomplete or contained syntax errors that prevented the generated roles from being executed. For Tomcat deployments, Claude Opus 4 achieved 95.2% compliance, failing one configuration check out of 21 CIS Level 1 requirements. Claude Sonnet 3.7 attained 85.7% compliance, missing three configuration items. Gemini 2.5 Pro and Pixtral Large achieved partial compliance at 38.1% and 47.1% respectively. For MongoDB deployments, Claude Opus 4 achieved 100% compliance, implementing all 7 CIS Level 1 requirements. Claude Sonnet 3.7, Gemini 2.5 Pro, and Pixtral Large each achieved 85.71% compliance, failing one control out of seven. The failed control was database file permissions.

The variation in compliance performance between technologies correlates with the number of controls. MongoDB's 7 controls yielded higher compliance rates across models compared to Tomcat's 21 controls. Claude Opus 4 maintained performance across both technologies, while Gemini 2.5 Pro improved from 38.1% (Tomcat) to 85.71% (MongoDB). This improvement in compliance from Tomcat to MongoDB occurred because fewer controls need to be implemented correctly to achieve a higher compliance percentage.

The quality ranking of a model within a dataset is a separate measure from

compliance achievement. A model's quality ranking depends on how its score compares to other roles in the same dataset, including human-written roles. Gemini 2.5 Pro ranked 5th among 135 Tomcat roles but 27th among 143 MongoDB roles. This difference in ranking does not indicate that Gemini 2.5 Pro generated lower quality MongoDB code. Rather, the MongoDB dataset contained a higher proportion of human-written roles with strong quality scores, which positioned Gemini 2.5 Pro lower in the ranking. Despite this lower quality ranking in the MongoDB dataset, Gemini 2.5 Pro achieved higher MongoDB compliance than Tomcat compliance. This confirms that compliance achievement is determined by the number of controls a model must implement correctly, not by where the model ranks in quality relative to other roles in the dataset.

We analysed the 75% failure rate and found that the 12 models that failed produced syntactically incorrect or incomplete code at the generation stage. This indicates that these models failed to comprehend the prompt rather than lacking the ability to implement compliance requirements. The input prompt sizes ranged from 1,572 to 3,403 tokens, representing less than 3% of the smallest context window (128K tokens) among the evaluated models.[15] The failure therefore cannot be attributed to prompt length exceeding model capacity. The three models in the inappropriate standards group (Nova Pro, GPT o3, Llama 4) all failed, demonstrating that models citing non-applicable frameworks cannot process structured multi-constraint prompts regardless of baseline coding ability. This finding challenges conventional capability predictors. Claude Sonnet 4 achieved the highest SWE-bench score (80.2%) yet failed due to incomplete code and syntax errors. In contrast, Pixtral Large succeeded despite lacking benchmark data in popular assessment platforms shown in Table 7.2. Similarly, Llama 4 with 10M tokens failed while Claude Opus 4 with 200K tokens succeeded, establishing that architectural features cannot predict performance on multi-constraint instruction-following tasks.

[15]IBM Research, "Larger Context Window," https://research.ibm.com/blog/larger-context-window.

The CO-STAR framework functions as an instruction-following test that requires models to perform several tasks simultaneously. Models must parse a multi-layered prompt structure, understand the priority classification system (A and B designations), interpret attribute-specific syntax (such as `mode=A` and `template_files = B`), and apply constraints across both quality and compliance domains while following verification loops. The four models that succeeded (Claude Opus 4, Claude Sonnet 3.7, Gemini 2.5 Pro, Pixtral Large) are all closed-source proprietary systems. This prevents investigation into the underlying factors that contributed to their success, such as parameter count, training data composition, or instruction-following algorithms, without collaboration with model providers.

7.5 Discussion

This section discusses the findings from the baseline evaluation, structured prompting, and compliance generation experiments. It examines what these findings mean for the use of foundation models in infrastructure code generation and identifies the factors that influence code quality and compliance outcomes.

Foundation models generate infrastructure code by predicting the next token based on patterns in their training data [143]. The training datasets contain code of varying quality, and the training process does not distinguish between well-engineered implementations and those containing security flaws. As a result, foundation models treat both as equivalent patterns for prediction. The baseline evaluation confirmed this behaviour. All 16 models produced code containing anti-patterns such as hardcoded values, missing error handling, and non-idempotent tasks. Of the 16 models, 15 scored below the dataset mean under both evaluation policies, with only Claude Opus 4 exceeding the mean.

The baseline evaluation also showed that models differ in their awareness of coding standards. Seven models cited appropriate domain-specific standards

such as Ansible Best Practices and ISO/IEC 25010 when asked about the standards they followed. Six models reported following no standards. Three models cited inappropriate standards, with GPT o3 referencing PEP 8 for YAML-based Ansible code and Llama 4 citing generic YAML formatting standards. Models in the inappropriate standards group produced the lowest quality code, performing worse than models that claimed no standards adherence. This finding carries a practical implication for practitioners. Models that cite standards are not necessarily applying them correctly. In some cases, models reference frameworks that are irrelevant to the domain of the generated code, and this misapplication actively degrades quality rather than improving it.

Application of the CO-STAR framework improved quality scores for all 16 models under both technologies, with improvements ranging from 9.1% to 50.1% for Tomcat and from 5.1% to 35.7 % for MongoDB. These improvements demonstrate that foundation models possess capabilities that are not activated during default code generation but can be activated through structured prompting. The variation in improvement across models indicates that different models respond to structured prompting to different degrees, but all 16 models showed improvement under both technologies.

Compliance generation produced contrasting results. Claude Opus 4 achieved 95.2% CIS compliance for Tomcat and 100% for MongoDB. These scores substantially exceed the best human-written implementations, which achieved 23.8% of the 21 CIS Level 1 controls for Tomcat and 42.85% of the 7 CIS Level 1 controls for MongoDB. However, 75% of models failed to generate deployable code when presented with combined quality and compliance prompts. The 12 models that failed produced output that was incomplete or contained syntax errors preventing execution.

The causes of failure differ between human-written and AI-generated implementations. Among human-written roles, 95% of Tomcat roles and 92.9% of MongoDB roles failed compliance requirements. Manual verification showed that these failures resulted from knowledge gaps, where developers were either

unaware of compliance requirements or unable to translate them into correct code. Among AI-generated roles, failures stemmed from prompt comprehension limitations rather than lack of compliance knowledge. The AI success rate of 25% (4 of 16 models) compared to the human success rate of 5% for Tomcat and 7% for MongoDB provides evidence that a properly instructed model can implement compliance requirements more effectively than manual human implementation when provided with detailed specifications.

This finding illustrates that the ability to follow complex instructions is equally or more important than model architecture for achieving compliance. Structured prompts compensate for missing quality and standards knowledge, as the per-category results in Section 7.4.1.1 show, but they do not compensate for limits in instruction-following capability, as the 12 models that failed the combined prompts demonstrate .

7.5.1 Implications for Developers

The findings from this chapter carry several implications for developers working with AI-generated infrastructure code.

First, the baseline evaluation showed that foundation models do not generate compliant infrastructure code by default. Developers who use AI coding assistants without providing quality and compliance guidance should expect the generated code to contain security smells and to fail compliance requirements. This applies regardless of whether the model is accessed directly or through a coding assistant tool, as neither mode introduces quality guidance through system prompts.

Second, the extended CO-STAR prompts developed in this research provide a reusable method for developers to integrate quality and compliance requirements into code generation. These prompts can be used in two ways. They can be integrated into AI coding assistants as system prompts, or they can be used independently as user prompts when interacting with foundation models directly. In both cases, the prompts encode domain expertise from An-

sible best practices and CIS benchmarks, lowering the barrier for developers who may be unfamiliar with these requirements.

Third, the results showed that not all foundation models can process complex prompts containing simultaneous quality and compliance constraints. Only 4 of 16 models succeeded at this task. Developers should therefore verify that their chosen model can generate deployable code from structured prompts before relying on it for compliance-aware generation.

Fourth, the mapping between CIS controls and the IaC Quality Framework's categories enables developers to verify compliance readiness prior to deployment using quality scores alone. This provides a faster feedback loop than deploying code and running post-deployment compliance scans, allowing developers to identify and address compliance gaps during development rather than after deployment.

7.5.2 Limitations

This study establishes the feasibility of compliance-based code generation within the constraints of the experimental design. The evaluation examined two technologies, Apache Tomcat version 10 and MongoDB version 7, implementing CIS Level 1 controls using Ansible with zero-shot prompting. This scope limits the generalisability of the findings to the specific technologies, compliance framework, and prompting strategy used in the evaluation.

The dataset comprised Ansible Galaxy roles made available by the community, representing community-validated implementations. Findings demonstrating that AI models outperformed human-written code should therefore be interpreted against community-standard baselines rather than enterprise repository standards.

Model capabilities were measured at a single point in time (July–August 2025). The AI coding assistant landscape was surveyed in July 2025. Given the pace of development in foundation models and AI coding tools, new models and updated versions may have been released after the evaluation period that

could affect the results. This challenge applies to all empirical studies involving foundation models, as model capabilities evolve between the time of evaluation and the time of publication.

Some assumptions were made when mapping IaC Quality Framework dimensions to CIS compliance requirements. A high score on both the structure and security dimensions of the quality framework is assumed to indicate that the code is ready for CIS compliance. Post-deployment CIS-CAT Pro validation demonstrated a correlation between these quality dimensions and CIS compliance. However, this relationship has not been validated across multiple regulatory frameworks.

The remaining limitations are threats to validity rather than matters of scope. The first concerns the relationship between the prompts and the evaluation criteria. The CO-STAR quality directives encode attributes that the evaluation framework also measures, raising the concern that models were guided to optimise for the same criteria against which their outputs were assessed. Three design features bound this threat. First, the baseline generations used task-only prompts containing no quality guidance, so the finding that models produce insecure code by default is untouched by it. Second, for the CO-STAR conditions the overlap is partially by design, because the directives and the framework attributes derive from the same documented Ansible community standards, and the claim under test is that structured prompting improves quality as the framework defines it. For that claim the shared criteria are the treatment rather than a confound. Third, two measurement instruments independent of the framework corroborate the results. CIS-CAT Pro scans assess the post-deployment state of the machine rather than the text the framework scores, and the manual inspection maps observed practices to Ansible documentation and CWE entries rather than to framework attributes.

The prompt design also contains two internal controls. None of the ten negative attributes in the evaluation policy was prompted. A drafted class of discouraged directives covering these anti-patterns was deliberately excluded

from the executed prompts, so avoidance of hardcoded credentials, suspicious comments, deprecated modules, silently ignored errors, and unnecessary shell usage was left to the models and then measured, leaving every affected category with an evaluation criterion the prompt never mentions. In addition, models discriminated between mandatory and conditional directives rather than inflating everything the prompt named. Categories whose prompted attributes were predominantly conditional, such as Automation and Integration, showed the smallest improvements, which is consistent with genuine instruction following rather than indiscriminate optimisation towards mentioned attributes.

One asymmetry remains and should be read as such. The human-written baseline code was produced without access to the quality directives, so the comparison between CO-STAR outputs and human-written code is one of guidance-assisted generation against unassisted development. Findings that AI models outperform human-written code under structured prompting must be interpreted with that asymmetry in mind.

A related threat concerns output stochasticity. Foundation models can produce different outputs for the same prompt across repeated runs, and each model and prompt configuration in this study was evaluated with a single generation, as stated in Section 7.2. The reported scores therefore carry no within-model variance estimate, and score differences between individual models should be interpreted with this uncertainty in mind. The conclusions of this chapter however rest on patterns rather than individual generations. The insecure-by-default finding holds across 15 of 16 models, the CO-STAR improvement is consistent across all 16 models in both technologies, and the per-category pattern replicates across two technologies. For stochastic variation to overturn these findings it would need to act in the same direction across 16 models and two technologies simultaneously. The cost factors that constrained the study to a single generation per configuration are described in Section 7.5.3, where variance quantification is identified as future work.

A final threat concerns training-data contamination. The human-written

baseline consists of public Ansible Galaxy roles that predate the training cut-offs of the evaluated models, so some of these roles were plausibly included in the training corpora, and no measures could rule this out. The direction of the resulting bias however works against the headline findings rather than for them. Contamination would push generated code towards the patterns of the community corpus, inflating baseline AI scores towards the human dataset mean, yet 15 of 16 models scored below that mean at baseline. The insecure-by-default finding is therefore conservative under contamination. Three further observations bound the threat. The task prompts specify functional requirements and never name repositories, authors, or Galaxy identifiers, so no model was directed towards any dataset item. The compliance results are validated by CIS-CAT Pro scans of deployed systems, which measure runtime behaviour rather than textual similarity to any training example. Finally, memorisation of community roles could not explain the compliance gap in either direction, because the best human-written implementations achieve 23.8% to 42.85% compliance while the successful models reach 85.7% to 100%, a level that reproduction of community code cannot produce. No decontamination audit was performed, because the training corpora of the evaluated models are closed and no reliable membership test exists for them. This remains an honest limitation of the study.

The directions arising from these constraints, including validation of the mapping between quality dimensions and compliance requirements across additional regulatory frameworks, are taken up in Section 7.5.3 .

7.5.3 Future Work

The findings from this chapter establish that structured prompting can improve the quality and compliance of AI-generated infrastructure code. Several directions for future work could build on these findings.

The evaluation in this chapter used zero-shot prompting, where each model received a single prompt without prior examples or iterative feedback. Fu-

ture work could compare this approach with multi-turn refinement, where the model receives feedback on its initial output and generates improved versions through successive interactions. This comparison would determine whether iterative prompting yields higher quality and compliance scores than the single-interaction approach used in this study.

The evaluation was conducted using a single generation per model and prompt configuration. Generations were produced manually in the web interfaces of the model providers under paid subscriptions, so each additional run would have required a further 96 manual sessions across the 16 models, two technologies, and three prompting conditions. The larger cost lies in validation. Verifying compliance required deploying each generated role to a separate test environment and running CIS-CAT Pro, which is a commercial tool requiring a paid licence. Each deployment involved provisioning a clean virtual machine, installing the target technology, executing the Ansible role, and running the CIS-CAT Pro scan. Running multiple generations per model would have multiplied this deployment and validation effort, which was not feasible within the budget and time available for this research. Repetition would also constitute a new study rather than a rerun, because provider-hosted models are updated continuously and the versions evaluated in July 2025 are no longer the versions served. Future work could perform repeated generations to determine whether model-level results , such as Claude Opus 4's 95–100% CIS compliance, represent consistent performance or vary between generations.

The evaluation examined two technologies using a single compliance framework (CIS Level 1) and a single IaC platform (Ansible). Future work could extend this evaluation to higher CIS implementation levels (IG2 and IG3), additional compliance frameworks (STIG, HIPAA, PCI-DSS), and other IaC platforms (Terraform, Puppet, Docker). Evaluating whether the approach generalises beyond configuration management to other IaC domains would demonstrate the broader applicability of the findings.

An alternative to structured prompting is fine-tuning, where compliance

knowledge is embedded directly into a model's weights through additional training rather than provided through the prompt at generation time. Low Rank Adaptation (LoRA) [144] is a technique that trains a small set of additional parameters (an adapter) that can be merged with a base model, allowing the model to be specialised for a specific task without retraining the full model. Future work could apply this technique in two stages. The first stage would train a LoRA adapter on a dataset of high-quality IaC code to embed code quality practices into an open-source foundation model. This adapter would then be merged with the base model. The second stage would train separate LoRA adapters for different compliance frameworks, such as CIS, STIG, and HIPAA, each encoding the requirements of a specific framework. Practitioners could then select the appropriate compliance adapter for their deployment context. This approach could address the instruction-following limitations observed in the 75% of models that failed to process complex prompts, as the compliance knowledge would be embedded in the model rather than requiring comprehension of a structured prompt. A comparative study between fine-tuning and prompting would help determine which approach produces more consistent compliance outcomes and which is more practical for integration into developer workflows.

Retrieval-augmented generation (RAG) offers a third position between these approaches, storing knowledge neither in the prompt nor in the model weights but in an external knowledge base queried at generation time. Quality directives, CIS controls, and other compliance requirements could be maintained in a structured store from which the relevant subset is retrieved for each task, rather than embedding all 51 quality directives and the compliance requirements statically in every prompt as this study did. This would improve the maintainability of the requirement set, because benchmark updates such as new CIS versions would require changing only the knowledge base, and the traceability of generated code, because each retrieved requirement can be linked to the tasks it informed. It could also improve consistency by ensuring

every generation draws on the same governed requirement set. The trade-off is methodological. Retrieval introduces additional architectural components whose behaviour varies between implementations, reintroducing exactly the attribution problem that the zero-shot single-interaction design of this study was chosen to avoid. A RAG-based comparison would therefore need to control for retrieval quality separately from generation quality. This constraint relaxes naturally in agentic settings. AI agents already operate through multi-step workflows, so an agent can consult the requirement store repeatedly during generation, retrieving the directives relevant to each subtask, verifying its output against them, and correcting failures before proceeding, rather than receiving the full requirement set in a single prompt. Extending this evaluation to agent-based generation backed by a maintained requirement store is therefore a natural continuation of this work. Whether retrieval-augmented generation, in single-shot or agentic form, matches or exceeds static structured prompting remains an open question for future work.

7.6 Chapter Summary

This chapter demonstrated that the IaC code quality framework can be extended beyond vulnerability detection to guide developers and AI models in producing compliant infrastructure code. Baseline evaluation of 16 foundational AI models generating 278 Ansible roles showed that 93.75% produce insecure code by default, scoring below dataset means with hardcoded values, missing error handling, and non-idempotent operations. Analysis identified three standards awareness groups. The appropriate standards group (7 models) achieved a mean score of 4.12, the no standards group (6 models) achieved 4.01, and the inappropriate standards group (3 models) achieved 3.65. These results establish that citing irrelevant guidelines produces worse outcomes than claiming no standards adherence. CO-STAR prompting improved quality scores for all 16 models, by 9.1–50.1% for Tomcat and 5.1–35.7% for

MongoDB, with structured prompting compensating for absent domain knowledge . The gains concentrated in the categories the prompt directives target, with Code Maintainability increasing most and Code Automation unchanged, while decreases were confined to individual model and category pairs . Only 25% of models succeeded at CIS Level 1 requirements, with failures correlating to instruction-following proficiency rather than coding benchmarks. Models achieving 80% on SWE-bench failed while others lacking benchmark data succeeded. Claude Opus 4 achieved 95–100% compliance versus 23–43% for human implementations. Four of the successful models are closed source, suggesting that compliance generation requires capabilities distinct from architectural characteristics. The framework integrates directly into existing systems through prompts and does not require any model retraining.

Chapter 8

Conclusion

Infrastructure as code has become a standard practice for managing and provisioning computing environments through machine-readable scripts rather than manual processes. As organisations increase their adoption of IaC tools for provisioning, configuration management, and deployment, the volume of infrastructure code in production environments has grown. This growth has introduced quality concerns. Infrastructure scripts that contain security vulnerabilities, misconfigurations, or non-idempotent operations can compromise entire deployment environments. Unlike application code, where defects may affect individual features, defects in infrastructure code can propagate across all systems managed by the affected scripts.

Despite these risks, no unified method existed for assessing the quality of infrastructure code. The survey of 24 static analysis tools conducted in Section 2.3.2 showed that existing tools focus on narrow aspects of code quality, such as syntax validation or specific security checks, and none incorporates metadata analysis into its quality metric calculations. Compliance frameworks such as CIS and STIG define security requirements for deployed systems, but no mechanism existed to verify whether infrastructure code satisfies these requirements prior to deployment. This fragmentation meant that organisations had no consistent way to measure, compare, or enforce quality standards across their infrastructure codebases.

This thesis addressed this gap by developing a framework for systematic quality assessment of infrastructure code and demonstrating its application across multiple use cases throughout the software development lifecycle. Figure 1.1 in Chapter 1 maps each identified challenge to the contribution addressing it and the chapter presenting it, and the paragraphs that follow revisit these contributions in turn.

The framework computes quality metrics across nine categories aligned with the principles outlined in ISO/IEC 25010:2011, incorporating metadata analysis as one of its core components (Chapter 3.2.3). As discussed in Section 1.3, the 2011 edition was superseded by ISO/IEC 25010:2023 during the course of our study. The 2023 revision does not introduce sub-characteristics that address infrastructure code concerns, and the gaps identified between the standard and IaC requirements persist across both editions. The framework and its findings therefore remain applicable under the revised standard. We conducted an empirical analysis of the framework's policy mechanism using data from 26,986 Ansible Galaxy repositories (Section 4.5). This analysis demonstrated the framework's ability to encode organisational priorities through configurable attribute and category weights across eight software development philosophies.

We applied the framework to security analysis, where it identified patterns invisible to traditional static analysis (Chapter 5). Supply chain vulnerability detection identified 45 dependency chains containing security smells affecting 117 repositories, with three repositories containing known CVEs. We applied security smell diffusion tracking (Chapter 6), which achieved 99.6–99.8% accuracy in detecting repository clones across platforms lacking fork metadata. This analysis showed that 38.4–54.1% of analysed repositories share significant code overlap, enabling vulnerability propagation across the ecosystem.

We evaluated AI-generated infrastructure code (Chapter 7) and found that 15 of the 16 foundational AI models tested produced code that scored below human-written baselines, with quality deficiencies persisting despite

high scores on standard benchmarks. Application of the extended CO-STAR prompting framework achieved 19–49% quality improvement across all models. When we integrated compliance requirements into the generation process, Claude Opus 4 achieved CIS benchmark compliance of 95–100%, compared to the highest human implementation which achieved 23–43%. These results demonstrate how the framework can be used throughout the Software Development Life Cycle (SDLC), from assessing existing repositories and identifying security threats, through tracking the propagation of smells, to assisting both human developers and AI systems in creating production-ready infrastructure code.

8.1 Future Work

While this thesis provides a validated framework for assessing IaC quality, expanding the scope and applicability of the framework presents an opportunity for future research. The framework has been implemented with a focus on configuration management tools, primarily Ansible, which limits the generalisability of the results to the wider IaC ecosystem. Extending the framework to include other tool types such as infrastructure provisioning and image building (e.g., Terraform [55] and Docker) will require modifications to the attribute mapping to accommodate different declarative syntax styles and methods of expressing resource dependencies. Prior work on IaC quality for other configuration management tools, such as Rahman's analysis of Puppet scripts [4], provides a foundation for identifying which quality attributes transfer across platforms and which require platform-specific definitions. Cross-platform quality metrics are also needed by organisations wishing to enforce consistent quality standards across their diverse IaC toolchains.

An additional area for future research is the integration of dynamic analysis techniques to complement the static analysis approach presented in this thesis. While static analysis can identify code-based quality issues, it does

not measure how well infrastructure performs at runtime. Aspects of runtime behaviour that are typically invisible to static analysis (e.g., performance characteristics, resource utilisation patterns) could be measured using dynamic analysis techniques.[1] The primary challenge of combining static and dynamic analysis lies in the cost and resources required to instrument and monitor infrastructure behaviour at scale. Continuous runtime analysis demands computational overhead and specialised expertise to implement and maintain, which limits its feasibility to large organisations with sufficient resources.

Our vulnerability detection work in the supply chain is based on repository metadata and static analysis to identify known patterns of vulnerabilities, but it may miss complex attack vectors that evade static analysis. Behavioural analysis techniques could complement this methodology by identifying anomalous patterns in repository evolution, author activity, and code modifications. Such techniques could detect attacks that manifest through subtle changes over time. Another approach is to use machine learning techniques trained on historical attack patterns. Ohm et al. [73] reviewed 20 publications focused on identifying and detecting software supply chain attacks, including machine learning methods for attack detection. Such techniques may be able to predict the likelihood of a vulnerability beyond the explicit security smells identified in the code.

Our investigation into AI code generation established the feasibility of producing production-ready infrastructure code through structured prompting. However, consistency remains an open challenge, as discussed in the limitations of Chapter 7. The fine-tuning approach described in Section 7.5.3 presents one direction for addressing these challenges by internalising quality and compliance requirements within model weights using techniques such as Low Rank Adaptation (LoRA) [144]. Frameworks for automated verification of generated code against quality and compliance requirements would also

[1] NCC Group, "Testing Infrastructure as Code Using Dynamic Tooling," 2022, https://www.nccgroup.com/research-blog/testing-infrastructure-as-code-using-dynamic-tooling/.

support integration with CI/CD pipelines. Whether multi-agent architectures employing domain-specific models for distinct components of infrastructure can improve quality and compliance outcomes remains an empirical question for future investigation.

Adopting the framework in production environments requires tooling that integrates with existing development workflows. As shown in Table 2.2, commercial IaC static analysis tools already provide DevOps integration, indicating that practitioners expect quality assessment to be embedded within their existing pipelines. Implementation as IDE plug-ins, CLI tools, and CI/CD pipeline components would lower adoption barriers while maintaining assessment consistency. Development of organisation-specific policy templates that encode company standards and compliance requirements would enable immediate use. Creating feedback mechanisms that learn from developer corrections and code review outcomes would continuously improve both assessment accuracy and generation quality over time.

References

[1] R. Wang, *Infrastructure as Code, Patterns and Practices: With examples in Python and Terraform.* ITpro collection, Manning, 2022.

[2] Center for Internet Security, "CIS Apache Tomcat v10 Benchmark." `https://www.cisecurity.org/benchmark/apache_tomcat`, February 2025.

[3] Center for Internet Security, "CIS Controls Version 8.1." `https://www.cisecurity.org/controls/v8`, March 2025.

[4] A. Rahman and L. Williams, "Source code properties of defective infrastructure as code scripts," *Information and Software Technology*, vol. 112, p. 148–163, Aug. 2019.

[5] S. D. Palma, D. D. Nucci, F. Palomba, and D. A. Tamburri, "Towards a catalogue of software quality metrics for infrastructure code," *ArXiv*, vol. abs/2005.13474, 2020.

[6] M. M. Hassan, J. Salvador, S. K. K. Santu, and A. Rahman, "State reconciliation defects in infrastructure as code," *Proc. ACM Softw. Eng.*, vol. 1, July 2024.

[7] P. A. Laplante, *What Every Engineer Should Know about Software Engineering (What Every Engineer Should Know).* USA: CRC Press, Inc., 2007.

[8] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 164–175, 2019.

[9] N. Borovits, I. Kumara, P. Krishnan, S. D. Palma, D. Di Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel, "Deepiac: Deep learning-based linguistic anti-pattern detection in iac," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, MaLTeSQuE 2020, (New York, NY, USA), p. 7–12, Association for Computing Machinery, 2020.

[10] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for puppet," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), p. 416–430, Association for Computing Machinery, 2016.

[11] A. Rahman, E. Farhana, C. Parnin, and L. Williams, "Gang of eight: A defect taxonomy for infrastructure as code scripts," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 752–764, 2020.

[12] International Organization for Standardization, "ISO/IEC 25010:2011 systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models." `https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en`, March 2011.

[13] International Organization for Standardization, "ISO/IEC 25010:2023 systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – product quality model." `https://www.iso.org/obp/ui/en/#iso:std:iso-iec:25010:ed-2:v1:en`, November 2023.

[14] International Organization for Standardization, "ISO/IEC 9126-1:2001 software engineering – product quality – part 1: Quality model." `https://www.iso.org/standard/22749.html`, June 2001.

[15] N. Saavedra and J. F. Ferreira, "GLITCH: Automated polyglot security smell detection in infrastructure as code." `https://arxiv.org/abs/2205.14371`, 2022.

[16] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, p. 1188–1221, 06 2018.

[17] P. R. R. Konala, V. Kumar, and D. Bainbridge, "Sok: Static configuration analysis in infrastructure as code scripts," in *IEEE International Conference on Cyber Security and Resilience, CSR 2023, Venice, Italy, July 31 - Aug. 2, 2023*, pp. 281–288, IEEE, 2023.

[18] B. Meijer, L. Hochstein, and R. Moser, *Ansible: Up and Running*. Sebastopol, CA: O'Reilly Media, 3rd edition ed., 2022.

[19] M. Chiari, M. De Pascalis, and M. Pradella, "Static analysis of infrastructure as code: a survey," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pp. 218–225, 2022.

[20] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code: An empirical study," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, p. 45–55, IEEE Press, 2015.

[21] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "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, 2019.

[22] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 189–200, 2016.

[23] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information and Software Technology*, vol. 108, pp. 65–77, 2019.

[24] I. Kumara, M. Garriga, A. U. Romeu, D. Di Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel, "The do's and don'ts of infrastructure code: A systematic gray literature review," *Information and Software Technology*, vol. 137, p. 106593, 2021.

[25] A. Brogi, A. Canciani, and J. Soldani, "Modelling and analysing cloud application management," in *Service Oriented and Cloud Computing* (S. Dustdar, F. Leymann, and M. Villari, eds.), (Cham), pp. 19–33, Springer International Publishing, 2015.

[26] Bridgecrew by Palo Alto Networks, "Checkov." `https://www.checkov.io/`, 2019.

[27] Aqua Security, "Cloudsploit." `https://cloudsploit.com/`, 2015.

[28] Chef Software, Inc., "Cookstyle." `https://github.com/chef/cookstyle`, 2016.

[29] J. Schwarz, A. Steffens, and H. Lichter, "Code smells in infrastructure as code," in *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pp. 220–228, 2018.

[30] J. Lepiller, R. Piskac, M. Schäf, and M. Santolucito, "Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities," in

Tools and Algorithms for the Construction and Analysis of Systems (J. F. Groote and K. G. Larsen, eds.), (Cham), pp. 105–123, Springer International Publishing, 2021.

[31] Checkmarx, "KICS: Keeping infrastructure as code secure." `https://www.kics.io/`, 2021.

[32] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Within-project defect prediction of infrastructure-as-code using product and process metrics," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2086–2104, 2022.

[33] K. Jayaraman, N. Bjørner, G. Outhred, and C. Kaufman, "Automated analysis and debugging of network connectivity policies," *Microsoft Research*, pp. 1–11, 2014.

[34] T. Dai, A. Karve, G. Koper, and S. Zeng, "Automatically detecting risky scripts in infrastructure code," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, (New York, NY, USA), p. 358–371, Association for Computing Machinery, 2020.

[35] Semgrep, Inc., "Semgrep." `https://semgrep.dev/`, 2020.

[36] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, 1 2021.

[37] Snyk, "SnykIaC: Infrastructure as code security." `https://snyk.io/product/infrastructure-as-code-security/`.

[38] I. Kumara, Z. Vasileiou, G. Meditskos, D. A. Tamburri, W.-J. Van Den Heuvel, A. Karakostas, S. Vrochidis, and I. Kompatsiaris, "Towards semantic detection of smells in cloud infrastructure code," in *Proceedings of the 10th International Conference on Web Intelligence, Mining and Semantics*, WIMS 2020, (New York, NY, USA), p. 63–67, Association for Computing Machinery, 2020.

[39] A. Brogi, A. Di Tommaso, and J. Soldani, "Sommelier: A tool for validating tosca application topologies," in *Model-Driven Engineering and Software Development* (L. F. Pires, S. Hammoudi, and B. Selic, eds.), (Cham), pp. 1–22, Springer International Publishing, 2018.

[40] Sonar, "SonarQube: Code quality and security." `https://www.sonarsource.com/products/sonarqube/`.

[41] M. M. Hassan and A. Rahman, "As code testing: Characterizing test quality in open source ansible development," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 208–219, 2022.

[42] Aqua Security, "tfsec." `https://github.com/aquasecurity/tfsec`. Version 1.28.1.

[43] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA, USA), Internet Society, 2005.

[44] T. Sutter, T. Kehrer, M. Rennhard, B. Tellenbach, and J. Klein, "Dynamic security analysis on android: A systematic literature review," *IEEE Access*, vol. 12, pp. 57261–57287, 2024.

[45] S. Winkler, *Terraform in Action*. Shelter Island, NY: Manning Publications, 2021.

[46] HashiCorp, "Terraform Registry." `https://registry.terraform.io/`, 2024.

[47] Microsoft Azure, "Azure Resource Manager (ARM) templates." `https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/`, 2024.

[48] Microsoft Azure, "Azure Quickstart Templates." `https://github.com/Azure/azure-quickstart-templates`, 2024.

[49] Google Cloud, "Google Cloud Deployment Manager." `https://cloud.google.com/deployment-manager`, 2024.

[50] D. Sandilands, *Puppet 8 for DevOpsEngineers: Automate Your Infrastructure at an Enterprise Scale*. Birmingham, UK: Packt Publishing, 2023.

[51] Puppet, "Puppet Forge." `https://forge.puppet.com/`, 2024.

[52] E. Waud, *Mastering Chef Provisioning*. Birmingham, UK: Packt Publishing, 2016.

[53] SaltStack, "SaltStack (Salt)." `https://saltproject.io/`, 2024.

[54] SaltStack, "SaltStack formula (Salt)." `https://github.com/saltstack-formulas`, 2024.

[55] M. Begoug, M. Chouchen, and A. Ouni, "TerraMetrics: An open source tool for Infrastructure-as-Code (IaC) quality metrics in Terraform," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, ICPC '24, (New York, NY, USA), p. 450–454, Association for Computing Machinery, 2024.

[56] B. Kitchenham and S. Pfleeger, "Software quality: the elusive target [special issues section]," *IEEE Software*, vol. 13, no. 1, pp. 12–21, 1996.

[57] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner, "Software quality models: Purposes, usage scenarios and requirements," in *2009 ICSE Workshop on Software Quality*, pp. 9–14, 2009.

[58] International Organization for Standardization and International Electrotechnical Commission, "ISO/IEC 9126:1991 software engineering — product quality," Dec. 1991. Withdrawn. ISO catalogue entry. Accessed: 2026-01-12.

[59] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code quality analysis in open source software development," *Information Systems Journal*, vol. 12, no. 1, pp. 43–60, 2002.

[60] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Plösch, A. Seidi, A. Goeb, and J. Streit, "The quamoco product quality modelling and assessment approach," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1133–1142, 2012.

[61] wibas GmbH, "Gp 2.1 establish an organizational policy," 2021. wibas CMMI Browser, Version 2.2.

[62] A. Sharma, R. Kumar, and P. S. Grover, "Estimation of quality for software components: an empirical approach," *SIGSOFT Softw. Eng. Notes*, vol. 33, p. 1–10, Oct. 2008.

[63] CMMI Product Team, "CMMI for development, version 1.3," Technical Report CMU/SEI-2010-TR-033, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2010.

[64] J. Taplin, *Move Fast and Break Things: How Facebook, Google, and Amazon Cornered Culture and Undermined Democracy*. New York, NY: Little, Brown and Company, 2017.

[65] Y. Qiao, Luqi, L. Zhang, and V. Berzins, "Documentation driven development for complex real-time systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 936–952, 2004.

[66] A. Stevens, *Enterprise Fortress: The Ultimate Handbook for Enterprise Security Architecture*. Sebastopol, CA, USA: O'Reilly Media, 1st ed., 2023.

[67] P. R. R. Konala, V. Kumar, D. Bainbridge, and J. Haseeb, "Metadata assisted supply-chain attack detection for ansible," in *Data and Applications Security and Privacy XXXIX* (S. Katsikas and B. Shafiq, eds.), (Cham), pp. 333–350, Springer Nature Switzerland, 2025.

[68] P. Nahta, "Securing the digital supply chain: Challenges, innovations, and best practices in cybersecurity," in *Innovation Management for a Resilient Digital Economy* (M. D. Lytras *et al.*, eds.), pp. 205–230, Hershey, PA: IGI Global, 2025.

[69] A. Ofori-Yeboah, R. Addo-Quaye, W. Oseni, P. Amorin, and C. Agangmikre, "Cyber supply chain security: A cost benefit analysis using net present value," in *2021 International Conference on Cyber Security and Internet of Things (ICSIoT)*, pp. 49–54, 2021.

[70] R. P. Kasturi, J. Fuller, Y. Sun, O. Chabklo, A. Rodriguez, J. Park, and B. Saltaformaggio, "Mistrust plugins you must: A Large-Scale study of malicious plugins in WordPress marketplaces," in *31st USENIX Security Symposium (USENIX Security 22)*, (Boston, MA), pp. 161–178, USENIX Association, Aug. 2022.

[71] M. J. Hossain Faruk, M. Tasnim, H. Shahriar, M. Valero, A. Rahman, and F. Wu, "Investigating novel approaches to defend software supply chain attacks," in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 283–288, 2022.

[72] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2021.

[73] M. Ohm and C. Stuke, "Sok: Practical detection of software supply chain attacks," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ARES '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[74] M. Kim, W. Jung, S. Lee, T. Kwon, and E. T. Kim, "Code repository vulnerability focusing on repojacking," in *2023 14th International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 1880–1884, 2023.

[75] P. R. R. Konala, V. Kumar, D. Bainbridge, and J. Haseeb, "Tracking security smell diffusion patterns in ansible playbooks using metadata," in *Advances in Information and Computer Security* (C. Cid and N. Yanai, eds.), (Singapore), pp. 371–390, Springer Nature Singapore, 2026.

[76] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Tech. Rep. 2007-541, School of Computing, Queen's University at Kingston, Ontario, Canada, September 2007.

[77] J. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '93, (Toronto, ON, Canada), pp. 171–183, IBM Press, 1993.

[78] J. Johnson, "Substring matching for clone detection and change tracking," in *Proceedings of the 10th International Conference on Software Maintenance*, ICSM '94, (Victoria, BC, Canada), pp. 120–126, IEEE Computer Society, 1994.

[79] U. Manber, "Finding similar files in a large file system," in *Proceedings of the Winter 1994 Usenix Technical Conference*, USENIX Winter '94, (San Francisco, CA, USA), pp. 1–10, USENIX Association, 1994.

[80] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of the 15th International Conference on Software Maintenance*, ICSM '99, (Oxford, UK), pp. 109–118, IEEE Computer Society, 1999.

[81] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. of 2nd Working Conference on Reverse Engineering (WCRE)*, pp. 86–95, 1995.

[82] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[83] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *Proc. of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 289–302, 2004.

[84] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proc. of*

the International Conference on Software Maintenance (ICSM), pp. 244–253, 1996.

[85] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. of the International Conference on Software Maintenance (ICSM)*, pp. 368–377, 1998.

[86] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. of the 29th International Conference on Software Engineering (ICSE)*, pp. 96–105, 2007.

[87] C. K. Roy and J. R. Cordy, "NiCad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. of the 16th IEEE International Conference on Program Comprehension (ICPC)*, pp. 172–181, 2008.

[88] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[89] T. Tsuru, T. Nakagawa, S. Matsumoto, Y. Higo, and S. Kusumoto, "Type-2 code clone detection for dockerfiles," in *2021 IEEE 15th International Workshop on Software Clones (IWSC)*, pp. 1–7, 2021.

[90] M. A. Oumaziz, J.-R. Falleri, X. Blanc, T. F. Bissyandé, and J. Klein, "Handling duplicates in dockerfiles families: Learning from experts," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 524–535, 2019.

[91] A. Rahman and C. Parnin, "Detecting and characterizing propagation of security weaknesses in puppet-based infrastructure management," *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3536–3553, 2023.

[92] J. Dean, O. Tofade, S. Poudel, and A. Rahman, "Propagation of insecure coding in configuration scripts." `https://publish.tntech.edu/index.php/PSRCI/article/view/688`, May 2020.

[93] Y. S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in git?: Use –histogram for code changes," *Empirical Software Engineering*, vol. 25, pp. 790–823, January 2020.

[94] P.-N. Tan, M. Steinbach, A. Karpatne, and V. Kumar, *Introduction to Data Mining*. Pearson, 2nd ed., 2019. Chapter 2 covers Jaccard coefficient as a similarity measure.

[95] J. Martinez-Gil, "Source code clone detection using unsupervised similarity measures," in *Software Quality as a Foundation for Security* (P. Bludau, R. Ramler, D. Winkler, and J. Bergsmann, eds.), (Cham), pp. 21–37, Springer Nature Switzerland, 2024.

[96] D. Reid, M. Jahanshahi, and A. Mockus, "The extent of orphan vulnerabilities from code reuse in open source software," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 2104–2115, Association for Computing Machinery, 2022.

[97] E. Larios Vargas, M. Aniche, C. Treude, M. Bruntink, and G. Gousios, "Selecting third-party libraries: the practitioners' perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, (New York, NY, USA), p. 245–256, Association for Computing Machinery, 2020.

[98] Google DeepMind, "Gemini cli." `https://ai.google.dev/gemini-api/docs/cli`, 2024. Command Line Interface for Gemini models.

[99] OpenAI, "Codex cli." `https://openai.com`, 2023. CLI interface for OpenAI Codex.

[100] P. Gauthier, "Aider." `https://aider.chat`, 2023. AI pair-programming in your terminal.

[101] Amazon Web Services, "Amazon q cli." `https://aws.amazon.com/q`, 2024. Developer assistant via AWS CLI.

[102] Anthropic, "Claude code." `https://claude.ai`, 2024. Code-oriented interface for Claude models.

[103] GitHub, "Copilot cli." `https://github.com/features/copilot`, 2023. GitHub Copilot command-line integration.

[104] Warp, "Warp 2.0." `https://warp.dev`, 2024. AI-powered terminal with modern features.

[105] C. A. Inc., "Cursor." `https://cursor.sh`, 2024. AI-powered code editor and CLI assistant.

[106] Replit, "Replit ghostwriter." `https://replit.com/site/ghostwriter`, 2023. AI coding assistant integrated in Replit.

[107] Codota, "Tabnine." https://www.tabnine.com, 2023. AI assistant for code completion and CLI use.

[108] StackBlitz Labs, "Bolt: Ai-powered full-stack web development in the browser." https://github.com/stackblitz-labs/bolt.diy, 2025.

[109] Meta AI, "Meta llama 4 system prompt." https://www.llama.com/docs/model-cards-and-prompt-formats/llama4/#-suggested-system-prompt-, 2024.

[110] SWE-bench Team, "Swe-bench leaderboards." https://www.swebench.com, 2025. Accessed: 2026-02-26.

[111] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," in *The Thirteenth International Conference on Learning Representations*, 2025.

[112] M. Eriksson, E. Purificato, A. Noroozian, J. Vinagre, G. Chaslot, E. Gomez, and D. Fernandez-Llorca, "Can We Trust AI Benchmarks? An Interdisciplinary Review of Current Issues in AI Evaluation." https://arxiv.org/abs/2502.06559, 2025.

[113] Z. Cheng, S. Wohnig, R. Gupta, S. Alam, T. Abdullahi, J. A. Ribeiro, C. Nielsen-Garcia, S. Mir, S. Li, J. Orender, S. A. Bahrainian, D. Kirste, A. Gokaslan, M. Glinka, C. Eickhoff, and R. Wolff, "Position: Benchmarking is Broken – Don't Let AI be Its Own Judge," in *Advances in Neural Information Processing Systems*, vol. 38, 2025.

[114] N. Tihanyi, T. Bisztray, M. A. Ferrag, R. Jain, and L. C. Cordeiro, "How secure is AI-generated code: a large-scale comparison of large language models." https://doi.org/10.1007/s10664-024-10590-1, 2025.

[115] J. Ji, J. Jun, M. Wu, and R. Gelles, "Cybersecurity risks of AI-generated code," *Center for Security and Emerging Technology*, 2024.

[116] European Parliament and Council of the European Union, "General Data Protection Regulation (GDPR)." https://gdpr-info.eu/, 2016.

[117] U.S. Department of Health and Human Services, "Health Insurance Portability and Accountability Act of 1996 (HIPAA)." https://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/index.html, 1996.

[118] U.S. Securities and Exchange Commission, "Sarbanes-Oxley Act of 2002." https://www.govinfo.gov/content/pkg/COMPS-1883/pdf/COMPS-1883.pdf, 2002.

[119] PCI Security Standards Council, "Payment Card Industry Data Security Standard (PCI-DSS)." https://www.pcisecuritystandards.org/pci_security.

[120] U.S. Congress, "Federal Information Security Modernization Act (FISMA)." https://www.congress.gov/bill/113th-congress/house-bill/1163.

[121] U.S. General Services Administration, "FedRAMP (Federal Risk and Authorization Management Program)." https://www.fedramp.gov.

[122] U.S. Department of Defense, "DoD Risk Management Framework (RMF)." https://public.cyber.mil/rmf.

[123] U.S. Department of Defense, "Cybersecurity Maturity Model Certification (CMMC)." https://dodcio.defense.gov/CMMC.

[124] SWIFT, "SWIFT Customer Security Programme (CSP)." https://www.swift.com/csp.

[125] New York Department of Financial Services, "New York Department of Financial Services Cybersecurity Regulation." https://www.dfs.ny.gov/industry_guidance/cybersecurity.

[126] StateRAMP, "StateRAMP." https://stateramp.org.

[127] UK National Cyber Security Centre, "UK Cyber Essentials Plus." https://www.ncsc.gov.uk/cyberessentials/overview.

[128] National Institute of Standards and Technology, "NIST Cybersecurity Framework (CSF)." https://www.nist.gov/cyberframework.

[129] HITRUST Alliance, "HITRUST Cybersecurity Framework (CSF)." https://hitrustalliance.net.

[130] Cloud Security Alliance, "Cloud Controls Matrix (CCM)." https://cloudsecurityalliance.org/research/ccm.

[131] Australian Prudential Regulation Authority, "APRA CPS 234 Information Security." https://www.apra.gov.au/cross-industry/prudential-standard-cps-234-information-security.

[132] North American Electric Reliability Corporation, "NERC Critical Infrastructure Protection (CIP)." https://www.nerc.com/pa/Stand/Pages/CIPStandards.aspx.

[133] U.S. Food and Drug Administration, "FDA Cybersecurity for Medical Devices." https://www.fda.gov/medical-devices/cybersecurity.

[134] U.S. Department of Education, CISA, and partners, "K-12 Cybersecurity Framework." https://www.cisa.gov/publication/k-12-cybersecurity-framework.

[135] Internal Revenue Service, "IRS Publication 1075: Tax Information Security Guidelines." https://www.irs.gov/privacy-disclosure/publication-1075.

[136] Federal Bureau of Investigation, "Criminal Justice Information Services (CJIS) Security Policy." https://www.fbi.gov/services/cjis/cjis-security-policy-resource-center.

[137] G. Falazi, U. Breitenbücher, F. Leymann, M. Stötzner, E. Ntentos, U. Zdun, M. Becker, and E. Heldwein, "On unifying the compliance management of applications based on iac automation," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pp. 226–229, IEEE, Mar. 2022.

[138] J. M. Imperial, M. D. Jones, and H. Tayyar Madabushi, "Standardizing intelligence: Aligning generative AI for regulatory and operational compliance." SSRN Electronic Journal, Jan. 2025.

[139] A. Verdet, L. D. Silva, A. Rahman, M. Openja, and F. Khomh, "Exploring security practices in infrastructure as code: An empirical study," *arXiv preprint arXiv:2308.03952*, 2023.

[140] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, (Red Hook, NY, USA), Curran Associates Inc., 2022.

[141] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: deliberate problem solving with large language models," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, (Red Hook, NY, USA), Curran Associates Inc., 2023.

[142] GovTech Singapore, "CO-STAR Framework." https://www.developer.tech.gov.sg/products/collections/data-science-and-artificial-intelligence/playbooks/prompt-engineering-playbook-beta-v3.pdf.

[143] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pp. 6000–6010, 2017.

[144] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *International Conference on Learning Representations (ICLR)*, 2022.

Appendices

Appendix A

Supplementary Figures

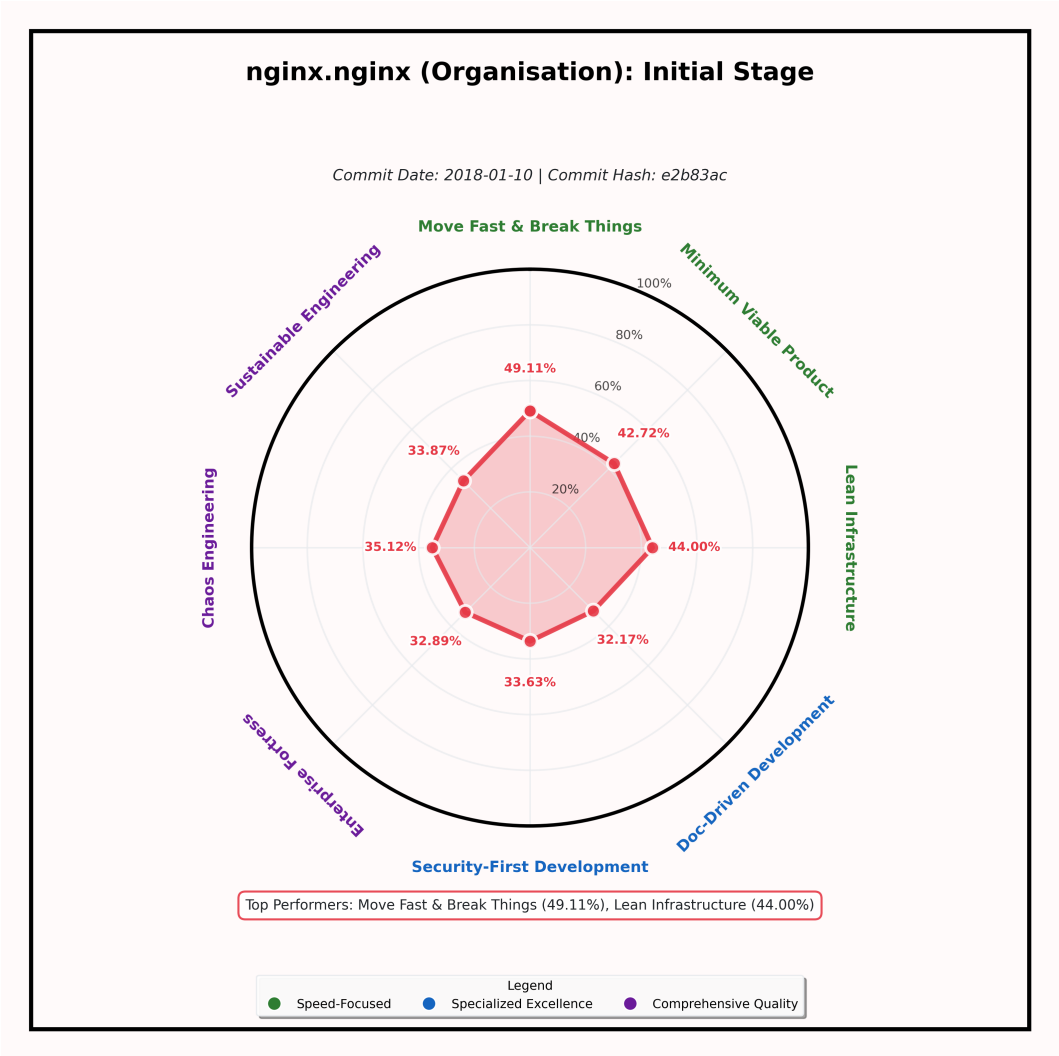


Figure A.1: Organisational Repository Philosophy Evolution: Official Nginx Repository, Stage 1 – Initial (enlarged from Figure 4.14)

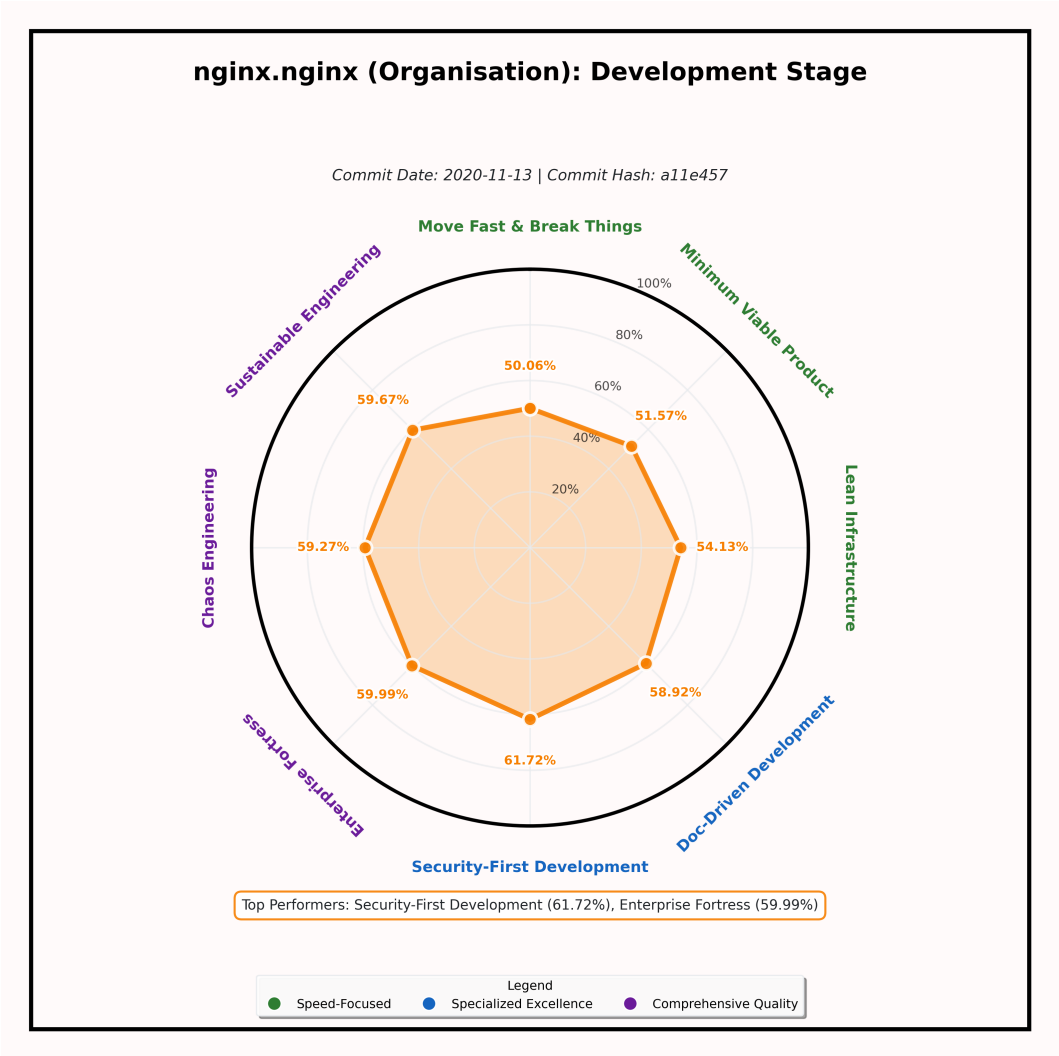


Figure A.2: Organisational Repository Philosophy Evolution: Official Nginx Repository, Stage 2 – Development (enlarged from Figure 4.14)

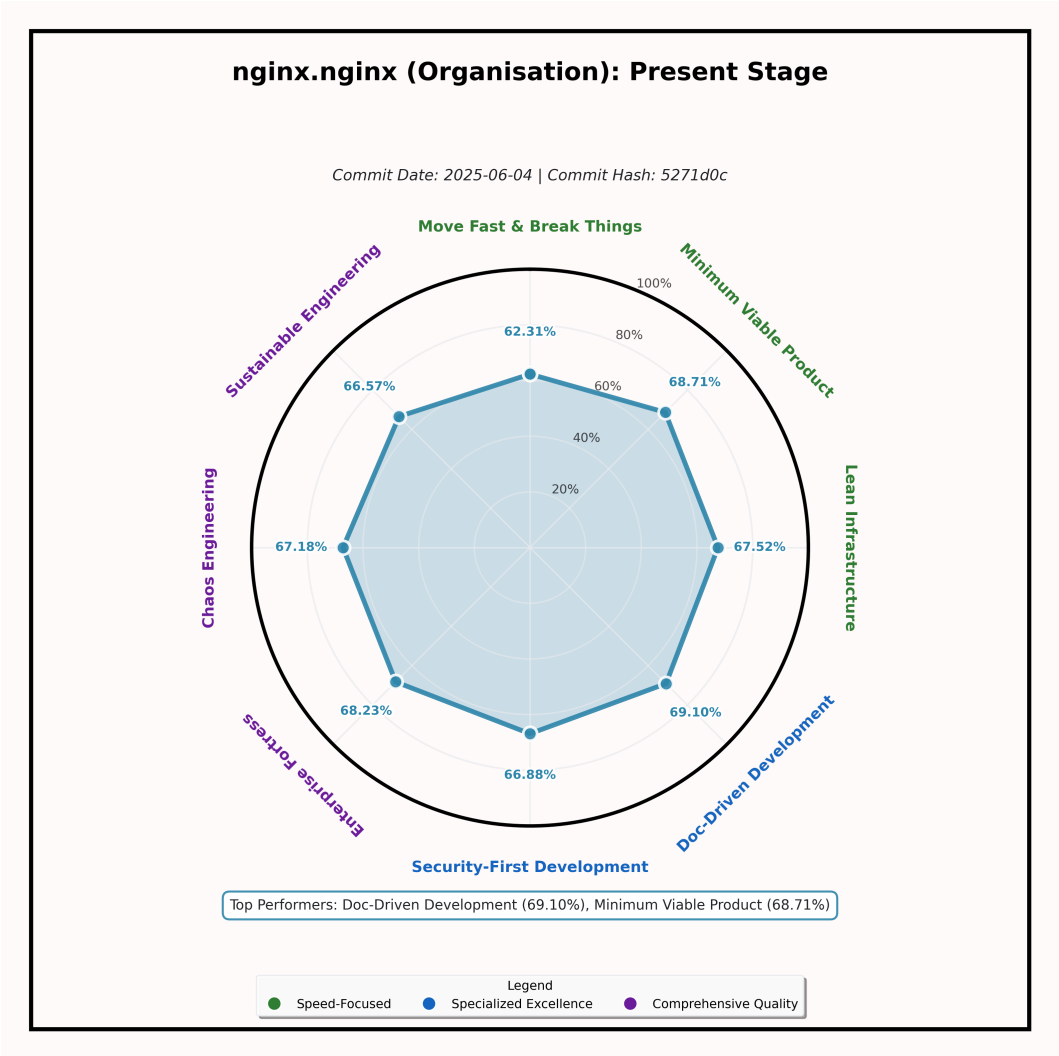


Figure A.3: Organisational Repository Philosophy Evolution: Official Nginx Repository, Stage 3 – Present (enlarged from Figure 4.14)

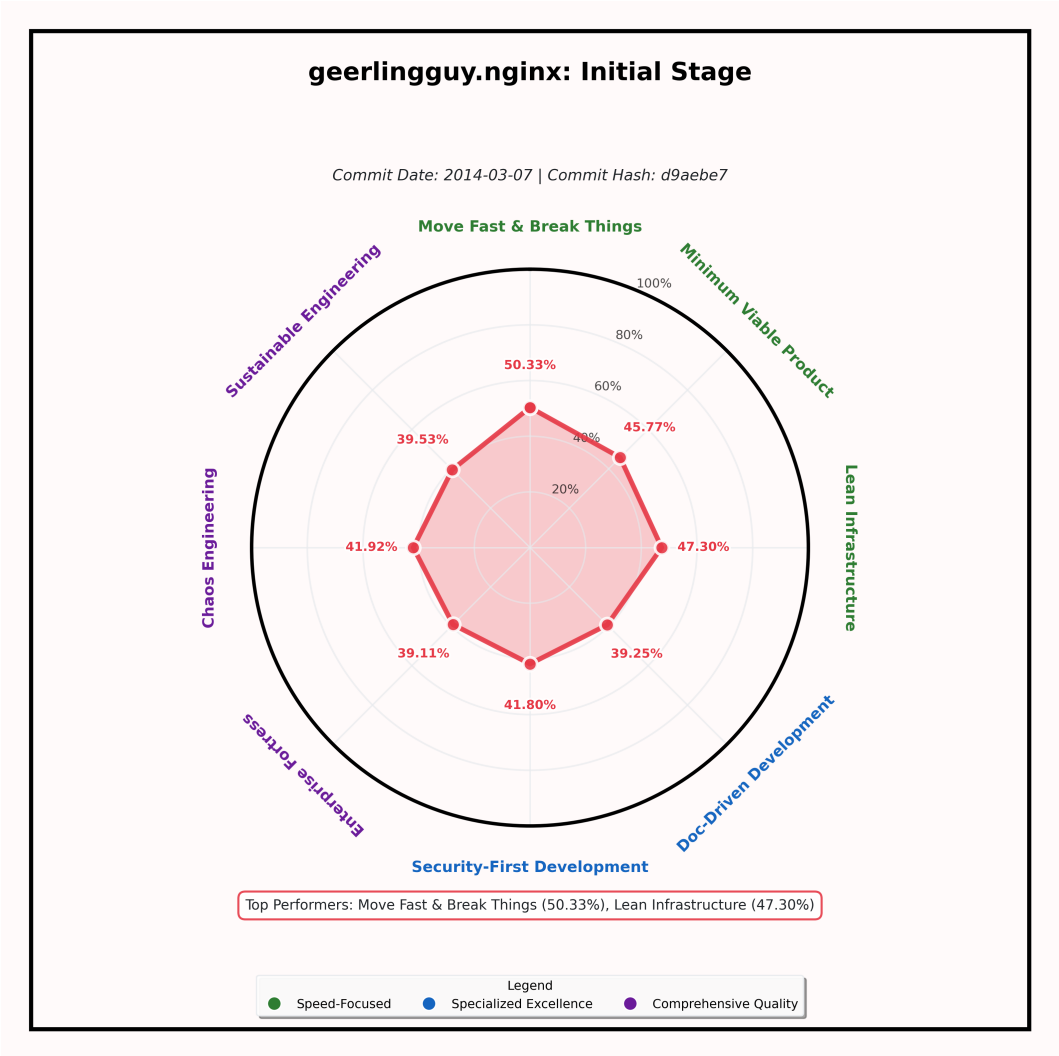


Figure A.4: Individual Repository Philosophy Evolution: `geerlingguy.nginx` Repository, Stage 1 – Initial (enlarged from Figure 4.16)

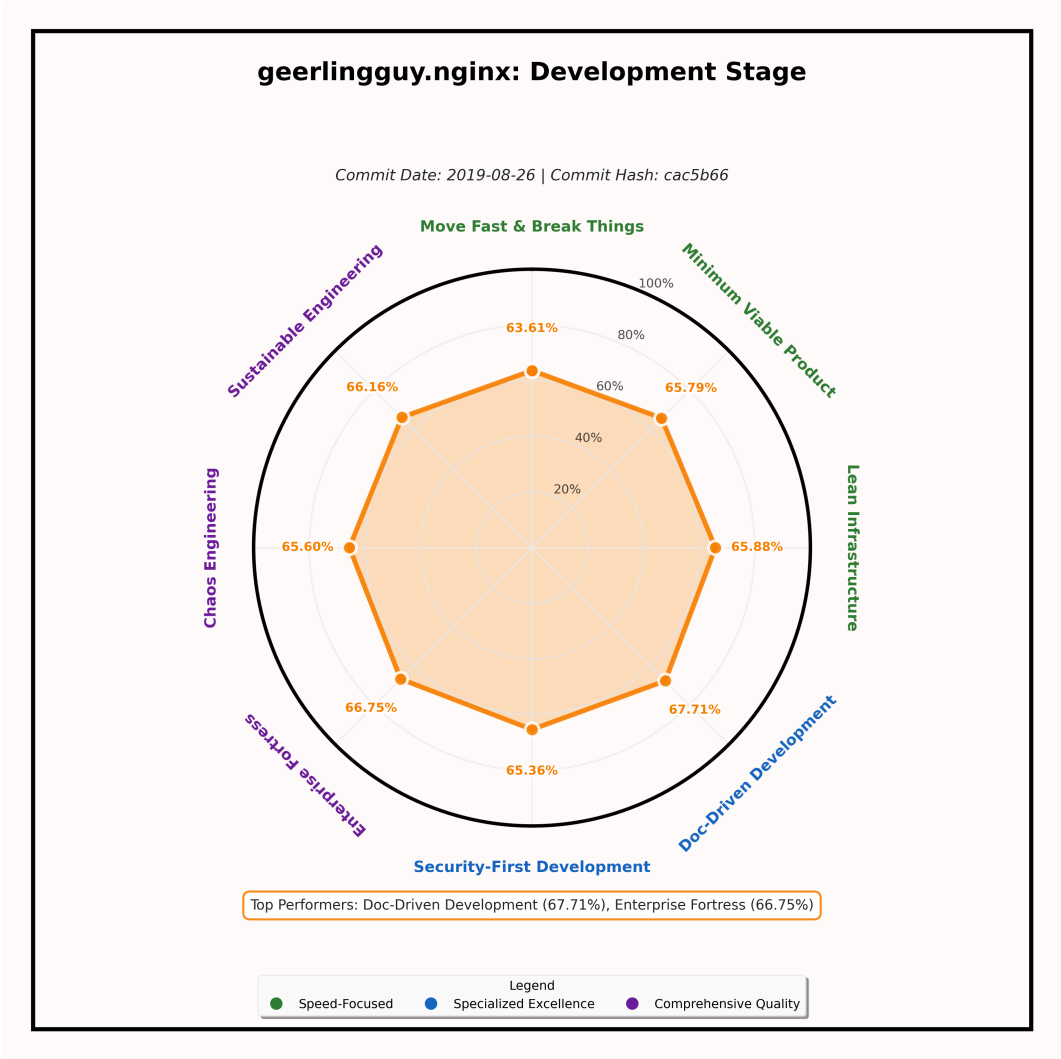


Figure A.5: Individual Repository Philosophy Evolution: `geerlingguy.nginx` Repository, Stage 2 – Development (enlarged from Figure 4.16)

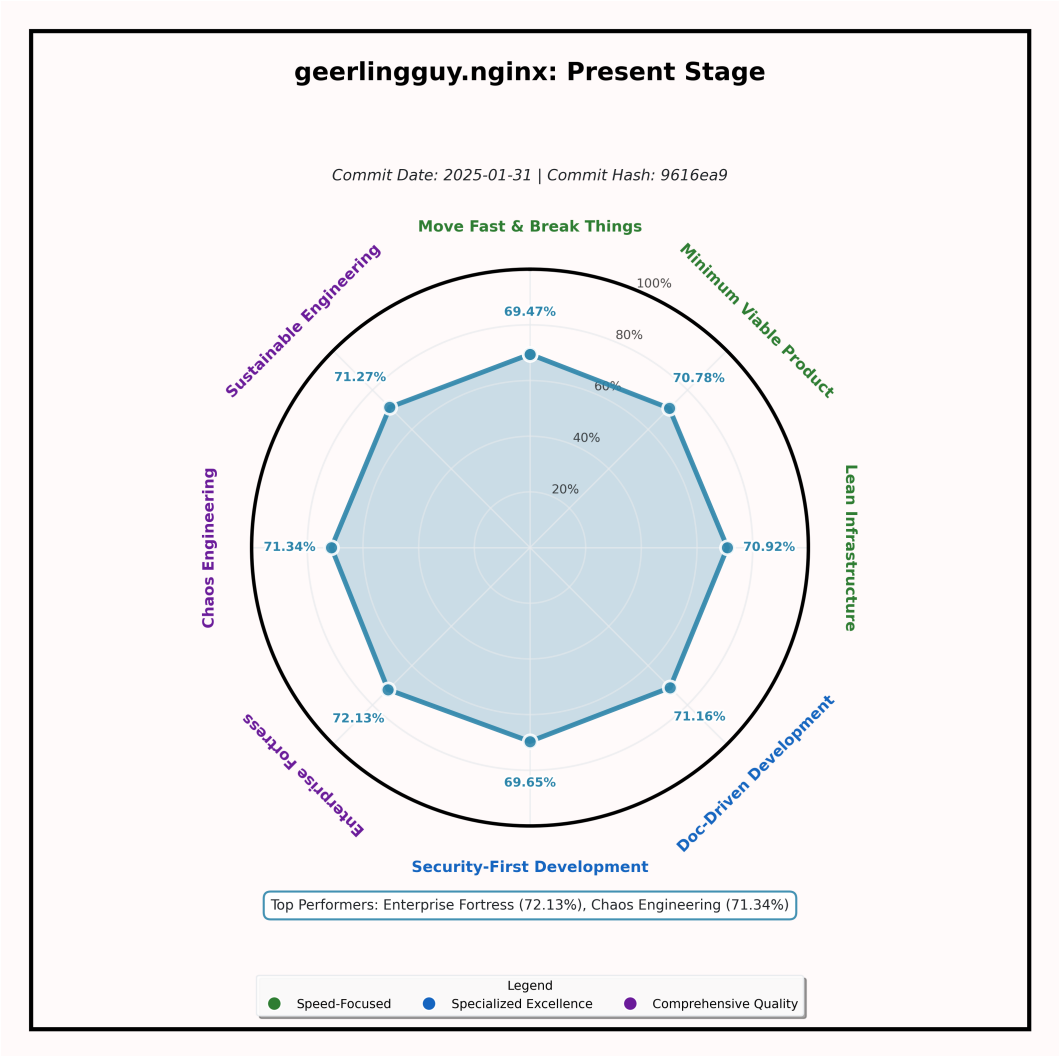


Figure A.6: Individual Repository Philosophy Evolution: `geerlingguy.nginx` Repository, Stage 3 – Present (enlarged from Figure 4.16)

Appendix B

CO-STAR Quality Directives

This appendix lists the complete set of 51 quality directives contained in the executed CO-STAR prompts of Chapter 7, of which the exemplar in Listing 7.3.1.1 shows two. The directive set comprises 25 mandatory directives (condition A) and 26 conditional directives (condition B) and was identical for both technologies, with only the OBJECTIVE line differing between the Tomcat and MongoDB sessions. No directive addresses the negative attributes of the evaluation policy, as anti-pattern avoidance was deliberately left unprompted.

Listing B: Quality Directives Contained In The Executed CO-STAR Prompts

```
MANDATORY DIRECTIVES (CONDITION A)
-comment_lines=A: Add detailed comments for EVERY task (no exceptions).
↪  Complex tasks need multiple comments. FAILURE WILL REQUIRE
↪  REGENERATION
-test_files=A: Include tests folder for role validation
-named_variable=A: Use descriptive variable names with role prefix
-named_task=A: Task should have a descriptive name
-conditionals=A: Use when conditions for OS-specific tasks
-value_conditionals=A: Validate variable values before use
-value_decisions=A: Make decisions based on variable values
-readme_file=A: Generate comprehensive README.md
-licence=A: Include proper licensing
-import_tasks / include_tasks=A: Modularize tasks using imports/includes
-vault=A: Use ansible-vault for where needed
-selinux=A: Handle SELinux contexts properly
-firewalld=A: Configure firewall rules for services
-mode=A: Set proper file permissions using mode parameter
-no_log=A: Use no_log for sensitive task output
-failed_when=A: Define custom failure conditions
-changed_when=A: Define when tasks report changes
-block/rescue/always=A: Use block error handling for critical tasks
```

```
-vars=A: Define all variables in appropriate locations
-check_mode=A: Support Ansible check mode (dry run)
-paths=A: Use proper path definitions
-wait_for=A: Wait for services to be ready before proceeding
-tag_count(metadata)=A: Include galaxy tag related to the objective in
↪ meta/main.yml
-platforms_count(metadata)=A: Define supported platforms in
↪ meta/main.yml
-min_ansible_version(metadata)=A: Specify minimum Ansible version in
↪ meta/main.yml
```

CONDITIONAL DIRECTIVES (CONDITION B)

```
-template_files=B: Use templates where configuration needs customization
-math_operations=B: Use arithmetic operations where needed
-regex=B: Use regex for pattern matching when appropriate
-lookups=B: Use lookup plugins for external data when beneficial
-user_interaction(input prompts)=B: Include vars_prompt for dynamic
↪ configuration
-ssh_authorized_key=B: Manage SSH keys if needed for access
-apt_key=B: Manage repository keys on Debian systems when needed
-debug=B: Include debug tasks for troubleshooting
-until/retry/delay=B: Use retry logic for unreliable operations
-any_errors_fatal=B: Stop on critical errors when appropriate in large
↪ (+5 hosts) deployments
-max_fail_percentage=B: Allow partial host failures in large (+5 hosts)
↪ deployments
-handlers=B: Use handlers for service restarts
-tags=B: Tag tasks for selective execution
-environment=B: Set environment variables when needed
-local_action=B: Run tasks locally when appropriate
-fetch=B: Retrieve files from remote hosts when needed
-URL/URI=B: Use uri module for API interactions when needed
-filters=B: Use for data transformation where needed
-fact_modules=B: Gather custom facts when beneficial
-become=B: Use privilege escalation when required for tasks
-synchtonize(rsync)=B: Use synchronize for large file transfers
-win_service=B: Support Windows services if needed
-add_host=B: Dynamically add hosts to inventory when needed
-git=B: Clone repositories when required
-import_role / include_role=B: Reuse other roles when appropriate
-include_vars=B: Load variables dynamically based on conditions
```