Evaluating and Preventing Security Smells in AI-Generated Ansible Code

Pandu Ranga Reddy Konala, Vimal Kumar, David Bainbridge, and Junaid Haseeb

School of Computing and Mathematical Sciences,
University of Waikato,
Hamilton, New Zealand 3240
{pkonala, vkumar, davidb, jhaseeb}@waikato.ac.nz

Abstract. AI coding assistants generate Infrastructure as Code, yet no work has examined whether this code meets security requirements. This matters because security smells in infrastructure code propagate to deployed systems, producing infrastructure that is insecure and untrustworthy. We evaluate 16 AI models generating Ansible roles for Apache Tomcat v10 and MongoDB v7, analysing 278 ansible roles against CIS benchmarks. Without security guidance, all 16 AI models produced code containing security smells, resulting in vulnerable infrastructure that fails compliance verification and underperforms code written by human developers. We introduce an approach integrating Ansible best practices and CIS benchmarks into prompts through an extended CO-STAR framework, enabling security smell prevention during synthesis rather than detection after deployment. When this approach is applied, 4 out of 16 models generate compliant code, with the leading model achieving 95%–100% CIS compliance, a fourfold improvement over humans at 23%–43%, with overall code quality improving by 19%–49%. The remaining 12 models fail not because they cannot generate code but because they cannot follow instructions with multiple constraints. For capable models, the approach requires no retraining and can be adopted through system prompts.

1 Introduction

Infrastructure as Code (IaC) automates system configuration through specifications that replace manual provisioning with version-controlled deployments [17]. Configuration management tools including Ansible, Terraform, and Puppet translate these specifications into deployed infrastructure. This automation introduces security risks, however. Security smells in infrastructure code, including hard-coded credentials, excessive permissions, and missing access controls, propagate directly to deployed systems, with the resulting infrastructure insecure and untrustworthy. Such infrastructure exposes organisations to data breaches, service disruptions, and regulatory penalties. In regulated industries, production deploy-

ments must satisfy compliance frameworks such as CIS Benchmarks[1] and DISA STIGs.[2] Code containing security smells fails compliance verification and cannot receive deployment authorisation.

AI coding assistants compound these security risks. Tools such as Cursor,[3] Bolt[4] and others generate IaC from natural language descriptions using foundational AI models such as OpenAI's GPT-4o and Anthropic's Claude Opus 4 as their knowledge bases. Studies report that 45%–62% of AI-generated code contains vulnerabilities [2,8]. It is also reported that IaC misconfigurations account for 68% of cloud security incidents [3]. Infrastructure misconfigurations can pose a greater security threat than application vulnerabilities because they become attack vectors immediately upon deployment.

Existing approaches to IaC security rely on detection rather than prevention, with static analysis tools identifying security smells after code enters repositories [20]. This detection paradigm is inadequate for AI-generated code because each refinement cycle incurs time and computational costs while requiring security expertise for remediation. More fundamentally, detection cannot prevent security smells from entering codebases. Research on IaC compliance remains limited to frameworks without validation [5,10], while studies on AI code security focus on application code rather than infrastructure [12,22]. No known work has examined the security properties of AI-generated infrastructure code or whether security requirements can be embedded into the code generation process.

We investigate two questions: (**RQ1**) *What is the default security posture of AI-generated infrastructure code and does it provide a viable foundation for achieving regulatory compliance?* (**RQ2**) *Can security-compliance requirements be embedded into the generation process itself, eliminating the need for iterative post-generation remediation?* We evaluate 16 foundational AI models generating Ansible code for Apache Tomcat and MongoDB deployments, assessing 278 roles against CIS benchmarks through both pre-deployment code quality analysis and post-deployment compliance verification. Our contributions are:

- We evaluated 16 foundational AI models generating Ansible code, quantifying default security behaviour and standards awareness across open-source and closed-source implementations.
- We extended the CO-STAR prompting framework to integrate Ansible best practices[5] and CIS benchmarks as generation constraints, enabling security smell prevention during synthesis for capable models without AI model retraining.

[1] Center for Internet Security, "CIS Critical Security Controls, Version 8," 2024, https://www.cisecurity.org/controls/cis-controls-list/.

[2] Defense Information Systems Agency, "Security Technical Implementation Guide (STIG)," 2025, https://public.cyber.mil/stigs/downloads/.

[3] Cursor AI Inc., "Cursor," 2024, https://cursor.sh.

[4] StackBlitz Labs, "Bolt: AI-Powered Full-Stack Web Development in the Browser," 2025, https://github.com/stackblitz-labs/bolt.diy.

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

2 Background

Production deployments in regulated industries must satisfy compliance frameworks that establish security objectives for data protection and system hardening. These frameworks specify requirements without prescribing implementations, requiring organisations to translate security objectives into configurations including file permissions, authentication mechanisms, and encryption protocols. Technical implementation guides bridge this gap by providing technology-specific security guidelines.

The CIS publishes benchmarks for operating systems, databases, web servers, and cloud platforms. Each benchmark organises security controls into three Implementation Groups. IG1 defines basic cyber hygiene forming the minimum security baseline, while IG2 and IG3 add controls for environments facing sophisticated threats. Each control specifies security requirements, audit procedures, and remediation steps.

The DISA provides Security Technical Implementation Guides (STIGs) for government systems. While STIGs target federal deployments, CIS benchmarks achieve broader commercial adoption. U.S. federal frameworks such as FedRAMP [25] and DoD RMF [23] mandate STIG compliance, while commercial frameworks such as PCI-DSS [18] and HIPAA [24] accept CIS benchmarks as security implementation standards.

Compliance verification occurs through automated scanning tools. CIS-CAT Pro,[6] the official assessment tool for CIS benchmarks, performs security audits against benchmark controls, while SCAP Compliance Checker[7] provides equivalent functionality for STIGs. These tools operate post-deployment, assessing running systems rather than the IaC that produced them. IaC generating non-compliant deployments cannot receive production authorisation, establishing security verification as a deployment gate. However, compliance with a specific framework does not guarantee security. Compliance controls are most effective when applied to code that already maintains a secure foundation, as regulatory requirements presuppose baseline security hygiene. Code quality thus serves as a prerequisite for effective compliance implementation.

2.1 AI Code Generation

While compliance frameworks define security and other requirements, AI models increasingly determine how IaC is generated. AI models generate code through next-token prediction, trained on corpora composed of of public repositories [26]. These corpora contain code of varying security quality, from hardened production implementations to demonstration snippets with security smells [7]. Training objectives reward syntactic validity and functional correctness without evaluating

[6] Center for Internet Security, "CIS-CAT Pro: Official Configuration Assessment Tool for CIS Benchmarks," https://www.cisecurity.org/cybersecurity-tools/cis-cat-pro.

[7] Naval Information Warfare Center Atlantic, "SCAP Compliance Checker," https://www.niwcatlantic.navy.mil/Technology/SCAP/.

security properties, and models have limited mechanisms to distinguish secure configurations from IaC with security smells [1,4].

AI coding assistants such as Cursor [11] and Bolt [21] provide interfaces to these models, processing natural language prompts and returning generated code. These tools employ system prompts to define role, output format, and behavioural constraints [16,28]. Analysis of system prompts from open-source tools such as Bolt and Llama reveals no instructions regarding security practices. Models therefore tend to reproduce patterns from training data without reliably distinguishing secure configurations from vulnerable ones. This shifts the quality-compliance burden from tool to operator, requiring users to specify security requirements explicitly.

2.2 Related Work

Related work addresses compliance management frameworks, AI code security, and IaC security analysis, though none examines AI-generated infrastructure code for compliance. Compliance management frameworks for IaC remain theoretical. Falazi *et al.* [5] proposed a technology-agnostic compliance management framework addressing configuration drift, and Imperial *et al.* [10] examined generative AI alignment with regulatory frameworks, yet neither provides empirical validation. AI code security research demonstrates that models introduce vulnerabilities in practice. Tihanyi *et al.* [22] found 62% of 331,000 AI-generated C programs contained vulnerabilities and Ji *et al.* [12] reported 48% of AI-generated code contains MITRE CWE Top 25 weaknesses. However, these studies focus on application code security rather than infrastructure security; whether similar patterns manifest in IaC was not examined.

Independent of AI-based code generation, IaC security research reveals deficiencies in human-written implementations. Rahman *et al.* [19] identified seven security smell categories across 1,093 repositories. Similar studies employ detection-based approaches, identifying security violations after code enters repositories rather than preventing them during generation. Static security smell detection tools surveyed by Konala *et al.* [20] assume human developers who implement security corrections iteratively. For AI-generated code, this approach provides no mechanism to prevent security smells during synthesis. No prior empirical work was found that examined the security properties of AI-generated infrastructure code or its viability for achieving compliance.

3 Baseline Security Performance of AI Models

To address **RQ1**, we evaluated foundational AI models generating Ansible roles for Apache Tomcat deployment without explicit security guidance. These outputs were compared against human-written implementations from community repositories, establishing baseline security performance and identifying deficiencies in AI-generated code.

3.1 Evaluation Protocol

To measure default security behaviour, we evaluated 16 foundational AI models using identical zero-shot prompts. We assessed the outputs for security smells, and compared the results against 119 human-written Ansible roles from community repositories.

Model Selection. Table 1 presents the 16 AI models evaluated, spanning organisations including Anthropic, OpenAI, Google, Meta, and DeepSeek. The selection captures diversity across licensing models, context window sizes (128K to 10M tokens), and coding benchmark performance.[8][9] All models were evaluated with standalone capabilities, disabling web search and retrieval augmented generation to ensure fair comparison.

Table 1. 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.

Tasks & Evaluation Procedure: Each model received an identical zero-shot prompt without examples or security guidance. This evaluates whether AI models produce secure code by default. The interaction comprised three phases: (Q1) a generation request: "*Generate an Ansible role for Tomcat 10 that installs and configures Host Manager for web applications*"; (Q2) a self-assessment query: "*Does the generated code follow any ISO code quality standards or best coding practices? (Yes/No)*"; and (Q3) a specification request if Q2 indicated adherence.

Generated code underwent quality analysis through the IaC quality framework [13], which assesses infrastructure code across nine dimensions including security and structure. The framework uses configurable policies to evaluate code against technology-specific standards. For this evaluation, we applied the Ansible

[8] SWE-bench, https://www.swebench.com/.

[9] LiveCodeBench, https://livecodebench.github.io/.

best practices policy to measure AI-generated code against Ansible documentation guidelines. This framework uses quality scores to detect supply chain vulnerabilities [14] and hidden vulnerability propagation pathways [15] that lie beyond the scope of IaC static analysis tools. Our evaluation extended beyond code analysis to examine AI models' awareness of quality standards through Q2 and Q3 responses. These self-reported responses may not reflect actual knowledge, as models can hallucinate standards adherence. However, comparing claimed awareness against actual code quality reveals gaps in model capabilities.

Dataset Composition: The evaluation dataset comprises 135 Ansible roles for Apache Tomcat 10: 16 AI-generated and 119 human-written roles from Ansible Galaxy [10]. Tomcat was selected for two reasons: CIS Apache Tomcat 10 Benchmarks[11] containing 21 Level 1 security controls enable compliance assessment in subsequent sections, and Ansible Galaxy contains sufficient human-written implementations for comparison. The IaC quality framework produces scores on a 9-point scale based on Ansible documentation requirements.

3.2 Baseline Findings

Evaluation produced dataset means of 4.56 for total quality (9-point scale, higher indicating better quality) and 0.634 for security (0–1 scale, higher indicating fewer security smells). Among 16 AI models, only Claude Opus 4 (5.07) exceeded the quality mean and only Perplexity Sonar (0.669) exceeded the security mean, with no model exceeding both. As Table 2 shows, AI-generated code consistently underperformed human-written implementations.

Categorisation based on Q2 and Q3 responses reveals three groups. These categories reflect claimed standards awareness rather than verified knowledge; models may cite standards without genuine understanding. Seven models citing domain-specific standards (Ansible Best Practices, ISO/IEC 25010) achieved quality scores averaging 4.08 but security scores of 0.599. Six models reporting no standards adherence achieved lower quality (3.95) but higher security (0.618). Three models citing inappropriate standards (PEP 8 for YAML, Ansible) produced both the lowest quality (3.57) and security (0.586) scores, indicating that misaligned guidance from non-applicable frameworks correlates with lower security and overall code quality.

Manual inspection revealed consistent security smells linked to Common Weakness Enumerations (CWEs) across AI-generated code. No model implemented Ansible Vault for credential management, exposing hardcoded sensitive data (CWE-798).[12] Only Claude Opus 4 implemented error handling, with the remaining 15 models omitting this practice (CWE-755).[13] Six models failed to

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

[11] Center for Internet Security, "CIS Apache Tomcat v10 Benchmark," Version 1.1.0, February 2025, https://www.cisecurity.org/benchmark/apache_tomcat.

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

[13] MITRE, "CWE-755: Improper Handling of Exceptional Conditions," https://cwe.mitre.org/data/definitions/755.html.

Table 2. Security and Quality Performance Analysis by Standards Awareness Category

Category	Model Name	Q3 Answer	Security Score (SS)	Total Quality Score (QS)	Category Avg (SS, QS)
Appropriate Standards	**Claude Opus 4**	Ansible Best Practices	85 (0.6136) ↓	21 (5.07) ↑	0.599, 4.08
	GPT 4o	ISO/IEC 25010 & 29110, OWASP IaC Sec Guide	114 (0.6068) ↓	115 (4.06) ↓	
	DeepSeek V3	ISO/IEC 25010, ISO/IEC 27001 (Indirect)	122 (0.5944) ↓	117 (4.030) ↓	
	Grok 4	General ISO/IEC 25010 Guidelines	116 (0.6049) ↓	118 (4.01) ↓	
	Pixtral Large	Ansible Best Practices	124 (0.5494) ↓	128 (3.85) ↓	
	Claude Sonnet 3.7	General Software & Ansible Best Practices	84 (0.6142) ↓	129 (3.80) ↓	
	Qwen3-235B-A22B	Ansible Best Practices	110 (0.6105) ↓	130 (3.73) ↓	
No Standards	**Claude Sonnet 4**	✕	82 (0.6160) ↓	75 (4.45) ↓	0.618, 3.95
	GPT o4-mini	✕	114 (0.6068) ↓	116 (4.037) ↓	
	Kimi K2	✕	111 (0.6086) ↓	121 (3.99) ↓	
	Sonar	✕	34 (0.6685) ↑	127 (3.88) ↓	
	GPT 4.1	✕	111 (0.6086) ↓	131 (3.72) ↓	
	Gemini 2.5 Pro	✕	121 (0.5988) ↓	132 (3.65) ↓	
Inappropriate Standards	**Nova Pro**	Confidential	124 (0.5494) ↓	133 (3.61) ↓	0.586, 3.57
	GPT o3	PEP 8 for Ansible	119 (0.6025) ↓	134 (3.58) ↓	
	Llama 4	YAML standards for Ansible	116 (0.6049) ↓	135 (3.51) ↓	

Legend: Rank (Score) format among 135 Tomcat roles; Security (0–1) and Quality (9-point) scores, higher is better; ↓ below mean, ↑ above mean. **Colour:** ■ Appropriate standards; ■ No standards (✕); ■ Inappropriate standards.

configure file permissions (CWE-276), namely Grok 4, Pixtral Large, GPT 4.1, Gemini 2.5 Pro, Nova Pro, and Llama 4.[14] No model included inline comments, and only 3 models (Claude Opus 4, Grok 4, and Sonar) included README files. Structural compliance was higher, with all models following directory conventions and 10 out of 16 producing syntax-error-free code. Grok 4, Qwen3-235B-A22B, GPT 4.1, Nova Pro, GPT o3, and Llama 4 contained syntax errors.

These findings address **RQ1**. The default security posture of AI-generated IaC is inadequate, with all 16 models producing code containing security smells and consistently underperforming human-written implementations. Structural correctness masks security violations including hardcoded credentials, missing error handling, and absent access controls. This baseline does not provide a foundation for compliance hardening, as regulatory frameworks presuppose security foundations that AI-generated code lacks by default. This inadequacy motivates **RQ2**, whether security-compliance requirements can be embedded into the generation process rather than remediated post-synthesis.

4 Methodology

To address RQ2, we present compliance-guided generation, an approach that embeds Ansible best practices and regulatory requirements as constraints during synthesis rather than detecting violations post-generation. Because baseline evaluation showed that models claim standards awareness without acting on it, our methodology makes the meaning of quality and compliance explicit rather than assuming the model already applies it. This requires mapping regulatory requirements to measurable code security attributes. The IaC code quality

[14] MITRE, "CWE-276: Incorrect Default Permissions," https://cwe.mitre.org/data/definitions/276.html.

framework [13], applied earlier in that evaluation, provides these measurements through repository-wide analysis. This investigation employs CIS Benchmarks rather than DISA STIGs, as CIS targets commercial systems across various industries, as established in Section 2, whereas STIGs focus on federal government deployments.

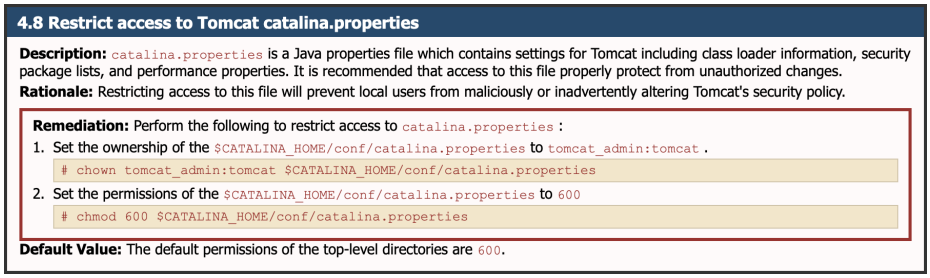

Fig. 1. Sample CIS Level 1 (IG1) Control from Apache Tomcat 10 Benchmark

Figure 1 presents a CIS Level 1 security control from the Apache Tomcat 10 Benchmark, illustrating how compliance requirements specify concrete technical security configurations. The remediation specifications show how requirements translate to code security attributes: `chown` and `chmod` commands specify ownership and permissions mapping to security attributes, while directory path specifications map to code structure attributes such as configuration templates.

We map CIS attributes to two dimensions of the IaC quality framework, which aligns with ISO/IEC 25010 (Software Quality Model). Structure scores serve as a primary filter: technology-specific compliance requirements demand configuration template files (e.g., `catalina.properties` for Tomcat, `mongod.conf` for MongoDB) where security controls are implemented. Without these templates, security controls cannot be applied, regardless of other code properties. Code security scores validate content within these templates, as compliance requirements specify permission values and configuration strings mapping to security attributes. Figure 2 illustrates this mapping from CIS security controls to framework categories. Roles achieving high scores in both dimensions indicate compliance readiness. Post-deployment validation through CIS-CAT Pro supports correlation between framework-based assessment and regulatory benchmark outcomes.

4.1 The Extended CO-STAR Prompting Framework

With assessment mechanisms established, we require a method to guide AI models toward generating quality-compliant code. The CO-STAR framework [6] structures generation instructions through six components: **C**ontext defines model role and expertise domain, **O**bjective specifies task requirements, **S**tyle establishes code formatting conventions, **T**one determines documentation characteristics, **A**udience identifies target users, and **R**esponse defines expected output format. We adopt this framework for a zero-shot approach to generating compliant code. Unlike Chain-of-Thought [27] or Tree-of-Thoughts [29] requiring

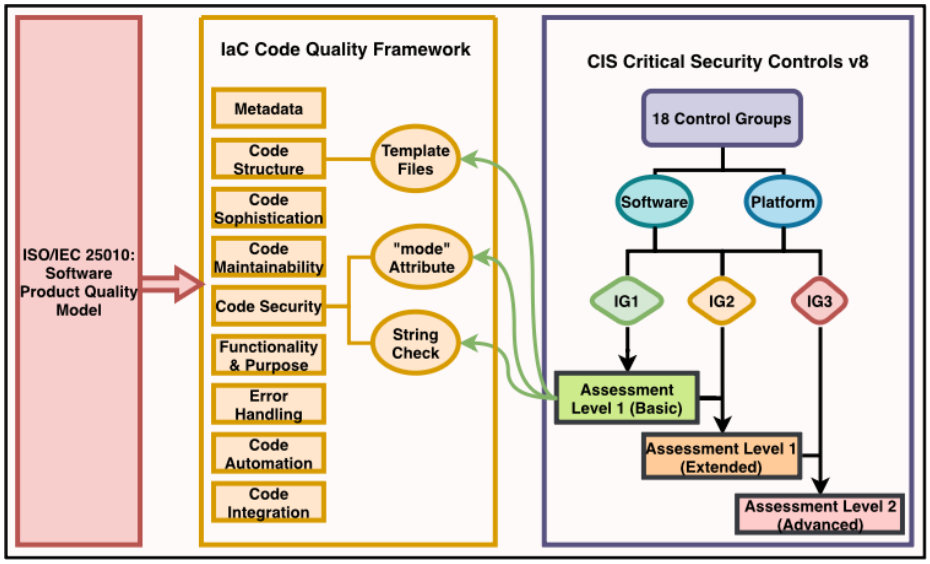

Fig. 2. Alignment of CIS Critical Security Controls (Level 1) with IaC Code Quality Framework Components

multiple interactions and API calls, or few-shot learning demanding curated example sets, zero-shot generation requires only a single interaction. This approach reduces experimental variables from example selection or iterative refinement, enabling direct comparison across AI models for generalisable results.

We extended CO-STAR with two categories for adding security requirements. The QUALITY category embeds Ansible best practices such as error handling and credential management through Ansible Vault. The COMPLIANCE category embeds CIS security controls specifying service configurations and access restrictions. Listing 1 demonstrates this structure. Each requirement specifies an attribute type (`mode` for permissions, `string` for configuration values, `readme_file` for documentation), an objective from Ansible best practices or CIS documentation, and an implementation directive.

Listing 1: Sample of Extended CO-STAR Prompt

```
1:
2:  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.
3:  ...
4:  STEP 1: [INSTRUCTIONS]
5:  Read and understand all instructions below carefully before proceeding.
6:  OBJECTIVE: Give me an ansible role for tomcat which installs and configures Host Manager for webapps.
7:  STYLE: Professional, avoid feature bloat, use bare minimum code, and stay LEAN - prioritise clarity over
        comprehensiveness.
8:  TONE: Clear and technical with comprehensive examples
9:  AUDIENCE: DevOps engineers, system administrators and interns
10: RESPONSE: Complete Ansible playbook structure with validation and testing
11: CONDITIONS: 'A' represents that it is a mandatory requirement, 'B' represents it is a recommended but context
        dependent. 'C' represents anti-patterns which should not be used.
12: STEP 2: [TECHNICAL REQUIREMENTS]
13: Generate the Ansible code strictly following these requirements based on their CONDITIONS values and avoid feature
        bloat. FAILURE WILL REQUIRE REGENERATION:
13:
14: ** COMPLIANCE:
15: -mode=A:Restrict access to Tomcat logs directory->Set the ownership of the $CATALINA_HOME/logs to tomcat_admin:tomcat
        and permissions to 0750.
16: -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
-password=C: Avoid usage of password
STEP 3: [VERIFICATION]
Recheck and confirm all TECHNICAL REQUIREMENTS are met. FAILURE WILL REQUIRE REGENERATION and return to STEP 1 and
        REDO.
```

Requirements use priority classification. Category 'A' denotes mandatory requirements from both Ansible best practices and CIS controls. Category 'B' denotes context-dependent practices. Category 'C' denotes anti-patterns to avoid, including hardcoded passwords and insecure defaults. The verification step instructs models to validate generated code against all requirements before completion.

4.2 Dataset Overview

To evaluate the extended CO-STAR framework, we expanded the dataset beyond Tomcat to include MongoDB, enabling investigation across technologies with varying security control complexity. Human-written roles were sourced from Ansible Galaxy, the official community repository providing community-validated implementations representing real-world deployment practices. The complete dataset comprises 278 Ansible roles:

- 135 Tomcat roles (16 AI-generated, 119 human-written) validated against 21 CIS Level 1 controls; input prompt comprises 3,403 tokens (1,696 words)
- 143 MongoDB roles (16 AI-generated, 127 human-written) assessed against 7 CIS Level 1 controls;[15] input prompt comprises 1,572 tokens (873 words)

MongoDB was selected for two reasons. Human-written ansible role counts (127) align with Tomcat (119), facilitating comparative analysis. Its 7 security controls contrast with Tomcat's 21, testing whether models maintain security accuracy as requirement count increases. This investigation focuses on CIS Level 1 controls establishing baseline security. The same 16 models from baseline evaluation were used. All roles were tested by deploying them in isolated test environments, with post-deployment validation performed using CIS-CAT Pro for assessing security compliance levels achieved.

5 Results and Discussion

Of the 16 evaluated models, only four generated syntactically correct code satisfying quality and compliance constraints: Claude Opus 4, Claude Sonnet 3.7, Gemini 2.5 Pro, and Pixtral Large. Compliance success correlated with standards awareness from baseline evaluation: three successful models came from the appropriate standards category, one from the no standards category, and none from the inappropriate standards category.

This 75% failure rate occurred despite input prompt sizes (1,572–3,403 tokens) representing less than 3% of the smallest context window (128K tokens)

[15] Center for Internet Security, "CIS MongoDB v7 Benchmark," Version 1.2.0, August 2025, https://www.cisecurity.org/benchmark/mongodb.

among evaluated models.[16] Claude Sonnet 4 achieved the highest SWE-bench score (80.2%) yet failed due to incomplete code and syntax errors, while Pixtral Large succeeded despite lacking benchmark data in popular assessment platforms shown in Table 1. Similarly, Llama 4 with 10M tokens failed while Claude Opus 4 with 200K tokens succeeded. These results suggest that instruction-following capability for multi-constraint security tasks likely emerges from training data rather than architectural features alone.

The extended CO-STAR framework functions as an instruction-following test requiring models to parse multi-layered prompt structure, understand categorical requirement systems (A/B/C priority classification), interpret attribute-specific syntax (`mode=A:`, `password=C:`), and apply constraints across quality and compliance domains. The four successful models are closed-source systems, preventing further investigation into underlying success factors without collaboration with model providers.

Table 3. AI Models Pre-deployment Quality Scores and Post-deployment Compliance

Model	Context Window	Structure/Security Score		CIS-CAT Compliance		Rank (Total Quality Score)	
		Tomcat	MongoDB	Tomcat	MongoDB	Tomcat	MongoDB
Claude Opus 4	200K	1.0/0.94	1.0/1.0	95.2%	100%	1 (6.03)	3 (5.61)
Claude Sonnet 3.7	200K	1.0/0.85	1.0/0.85	85.7%	85.7%	8 (5.38)	7 (5.37)
Pixtral Large	128K	0.9/0.62	1.0/0.85	47.1%	85.7%	37 (4.91)	50 (4.67)
Gemini 2.5 Pro	1M	0.9/0.57	1.0/0.85	38.1%	85.7%	5 (5.45)	27 (4.88)

Colour Coding: ■ = Appropriate standards; ■ = No standards

Table 3 presents context window sizes, pre-deployment structure/security scores, post-deployment CIS-CAT compliance outcomes, and total quality scores for the four successful models, demonstrating that pre-deployment quality metrics indicate regulatory conformance. Analysis reveals correlation patterns: higher structure/security scores correspond to increased compliance rates, while MongoDB's reduced control count (7 vs. Tomcat's 21) enabled higher compliance achievement across models. These results establish that structure and security metrics can enable pre-deployment verification of compliance readiness.

Human-written Ansible Galaxy roles also demonstrated compliance failures. Among 119 Tomcat roles, 113 (95.0%) lacked configuration templates for CIS security controls (structure score < 1), while the six roles with templates achieved maximum 23.8% compliance. For MongoDB, 118 of 127 roles (92.9%) similarly lacked templates (structure score < 1), with nine implementing roles achieving maximum 42.85% compliance. Failures across both technologies resulted from incorrect permission syntax, malformed configuration strings, and missing security-compliance implementations. Users adopting these human-written roles for regulated deployments face substantial remediation effort, as even the highest-performing implementations require correction of 57.1%–76.2% of controls before achieving compliance authorisation and quality improvement.

[16] IBM Research, "Larger Context Window," https://research.ibm.com/blog/larger-context-window.

Beyond compliance, successful AI-generated roles achieved competitive quality rankings among all 278 implementations. Claude Opus 4 ranked 1st and 3rd (6.03, 5.61), Claude Sonnet 3.7 8th and 7th (5.38, 5.37), Gemini 2.5 Pro 5th and 27th (5.45, 4.88), and Pixtral Large 37th and 50th (4.91, 4.67) for Tomcat and MongoDB respectively. Manual inspection of these roles found none of the security smells identified during baseline analysis. All four models implemented Ansible Vault for credential management and included proper file permissions. Compared to baseline evaluation where models received no security guidance (Table 2), quality scores improved by 19%–49%. Claude Opus 4 improved from 5.07 to 6.03 (19%), Pixtral Large from 3.85 to 4.91(28%), Claude Sonnet 3.7 from 3.80 to 5.38(42%), and Gemini 2.5 Pro from 3.65 to 5.45(49%). For the four successful models, structured prompting with quality and compliance constraints produced code free of security smells, providing a strong foundation for compliance whilst achieving higher code quality than human-written implementations.

For capable models, quality-compliance requirements can be embedded into code generation, eliminating iterative post-generation remediation. The leading model achieved 95.2%–100% CIS compliance through single-generation structured prompting compared to 23%–43% for human-written code. The extended CO-STAR framework demonstrates that quality-compliance guided generation produces results that detection-based approaches cannot attain, addressing **RQ2**.

5.1 Prevention Versus Detection

Current approaches to IaC security rely on detection, with static analysis tools identifying security smells after code enters repositories. This paradigm is inadequate for AI-generated code because it cannot prevent security smells from being introduced during synthesis, instead requiring iterative remediation cycles that incur time and computational costs. Our findings support a prevention-based alternative grounded in the observation that code quality serves as a prerequisite for compliance. CIS controls assume that foundational code is already of high quality with proper security measures such as permissions and credential management in place, providing the base upon which compliance controls are applied. For capable models, embedding quality-compliance controls during synthesis produces compliant code rather than requiring post-hoc correction. The correlation between pre-deployment quality metrics and post-deployment CIS-CAT outcomes further validates this approach, enabling compliance verification before code reaches production environments. Prevention-based generation complements rather than replaces existing static analysis tools, as detection remains valuable for auditing committed code while quality-compliance guided code generation prevents violations from entering codebases in the first instance.

5.2 Implications for Practitioners

Compliance-aware generation improves program comprehension for developers and the IaC community. Our methodology serves as executable documentation,

with security requirements explicitly stating objectives alongside implementation directives, making security intent transparent in the codebase. Code generated by successful models exhibits consistent structure with configuration templates, proper directory organisation, and README files, reducing cognitive load when onboarding team members or auditing deployments. The mapping between CIS controls and code attributes provides traceability from security requirements to implementation, enabling developers to understand why specific configurations exist. This contrasts with human-written implementations where security rationale is rarely documented, forcing maintainers to reverse-engineer intent from configuration values. For the IaC community, our extended CO-STAR prompts establish reusable patterns encoding security expertise, lowering the barrier for practitioners unfamiliar with best practices and regulatory frameworks. These prompts integrate with AI models and coding assistants as system prompts or can function independently as user prompts.

5.3 Limitations and Future Work

Internal validity: Our evaluation captures AI models capabilities at a specific point in time; AI models evolve rapidly, and results may differ with newer versions which is a limitation common to empirical studies in this rapidly evolving domain. **External validity:** The evaluation focuses on two technologies (Apache Tomcat and MongoDB) with CIS Level 1 controls using Ansible and zero-shot interactions as a foundational study. Dataset composition leverages publicly available Ansible Galaxy roles representing community contributions that may differ from proprietary organisational deployments. **Construct validity:** The mapping from IaC quality framework dimensions to CIS compliance requirements assumes structure and security scores indicate compliance readiness; while post-deployment CIS-CAT validation supports this correlation, independent validation across diverse regulatory frameworks remains necessary.

In future work, we will expand this investigation to higher implementation groups (IG2, IG3), alternative compliance frameworks (STIG, PCI-DSS), diverse IaC technologies (Terraform, Puppet), and multi-turn refinement approaches. Comparative analysis of fine-tuning versus prompting would quantify whether embedding compliance knowledge through model training yields superior outcomes. Fine-tuning open-source models using Low-Rank Adaptation (LoRA) [9] on compliant IaC code could internalise security practices and compliance requirements, potentially addressing instruction-following limitations observed in the 75% of models that failed complex multi-constraint tasks.

6 Conclusion

This paper investigated whether AI models can generate quality-compliant IaC. Evaluation of 16 AI models and 278 Ansible roles demonstrated three findings. First, all 16 AI models produced code containing security smells by default, underperforming human-written implementations. Second, 12 out of 16 models failed not from coding inability but from instruction-following limitations with multiple constraints. Models achieving 80% on SWE-bench failed while

others lacking benchmark data succeeded. Third, structured prompting enables four models to generate compliant code, with the leading model achieving 95%–100% CIS compliance compared to 23%–43% for human implementations, with overall code quality improving by 19%–49%. Manual inspection confirmed the four successful roles contained no security smells previously detected, providing a strong foundation for compliance. All successful models are closed-source, indicating secure-compliant generation depends on capabilities emerging from training methodologies rather than architectural features alone. For capable models, our approach deploys immediately through system prompts without model retraining. Validation across higher implementation groups, alternative frameworks, and diverse IaC technologies remains necessary to establish generalisability.

7 Data Availability

The dataset and prompts of this study are available at: `https://figshare.com/s/67c36294c704c7ab5ae5`.

References

1. Cheng, Z., Wohnig, S., Gupta, R., Alam, S., Abdullahi, T., Ribeiro, J.A., Nielsen-Garcia, C., Mir, S., Li, S., Orender, J., Bahrainian, S.A., Kirste, D., Gokaslan, A., Glinka, M., Eickhoff, C., Wolff, R.: Position: Benchmarking is Broken – Don't Let AI be Its Own Judge. In: Advances in Neural Information Processing Systems. vol. 38 (2025), `https://neurips.cc/virtual/2025/poster/121919`
2. Cloud Security Alliance: Understanding security risks in AI-generated code. `https://cloudsecurityalliance.org/blog/2025/07/09/understanding-security-risks-in-ai-generated-code` (Jul 2025)
3. CloudPSO: Cloud security is failing in 2025 due to misconfigurations. `https://cloudpso.com/cloud-security-is-failing-in-2025-due-to-misconfigurations/`
4. Eriksson, M., Purificato, E., Noroozian, A., Vinagre, J., Chaslot, G., Gomez, E., Fernandez-Llorca, D.: Can We Trust AI Benchmarks? An Interdisciplinary Review of Current Issues in AI Evaluation. `https://arxiv.org/abs/2502.06559` (2025)
5. Falazi, G., Breitenbücher, U., Leymann, F., Stötzner, M., Ntentos, E., Zdun, U., Becker, M., Heldwein, E.: 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)
6. 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`
7. Hajipour, H., Hassler, K., Holz, T., Schönherr, L., Fritz, M.: Codelmsec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models. `https://arxiv.org/abs/2302.04012` (2023)
8. Help Net Security: AI can write your code, but nearly half of it may be insecure. `https://www.helpnetsecurity.com/2025/08/07/create-ai-code-security-risks/` (Aug 2025)
9. Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: Lora: Low-rank adaptation of large language models. In: International Conference on Learning Representations (ICLR) (2022)

10. Imperial, J.M., Jones, M.D., Tayyar Madabushi, H.: Standardizing intelligence: Aligning generative ai for regulatory and operational compliance. SSRN Electronic Journal (Jan 2025). https://doi.org/10.2139/ssrn.5146161
11. Inc., C.A.: Cursor. https://cursor.sh (2024), AI-powered assistant
12. Ji, J., Jun, J., Wu, M., Gelles, R.: Cybersecurity risks of AI-generated code. https://cset.georgetown.edu/wp-content/uploads/CSET-Cybersecurity-Risks-of-AI-Generated-Code.pdf (2024)
13. Konala, P.R.R., Kumar, V., Bainbridge, D., Haseeb, J.: A framework for measuring the quality of infrastructure-as-code scripts. https://arxiv.org/abs/2502.03127 (2025)
14. Konala, P.R.R., Kumar, V., Bainbridge, D., Haseeb, J.: Metadata assisted supply-chain attack detection for ansible. In: Katsikas, S., Shafiq, B. (eds.) Data and Applications Security and Privacy XXXIX. pp. 333–350. Springer Nature Switzerland, Cham (2025)
15. Konala, P.R.R., Kumar, V., Bainbridge, D., Haseeb, J.: Tracking security smell diffusion patterns in ansible playbooks using metadata. In: Cid, C., Yanai, N. (eds.) Advances in Information and Computer Security. pp. 371–390. Springer Nature Singapore, Singapore (2026)
16. Meta AI: Meta llama 4 system prompt. https://www.llama.com/docs/model-cards-and-prompt-formats/llama4/#-suggested-system-prompt-
17. Morris, K.: Infrastructure as Code. O'Reilly (2021)
18. PCI Security Standards Council: Payment Card Industry Data Security Standard (PCI-DSS). https://www.pcisecuritystandards.org/pci_security
19. Rahman, A., Parnin, C., Williams, L.: The seven sins: Security smells in infrastructure as code scripts. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 164–175. IEEE (2019)
20. Reddy Konala, P.R., Kumar, V., Bainbridge, D.: SoK: Static configuration analysis in infrastructure as code scripts. In: 2023 IEEE International Conference on Cyber Security and Resilience (CSR). pp. 281–288 (2023)
21. StackBlitz Labs: Bolt: AI-powered full-stack web development in the browser. https://github.com/stackblitz-labs/bolt.diy (2025)
22. Tihanyi, N., Bisztray, T., Ferrag, M.A., Jain, R., Cordeiro, L.C.: How secure is AI-generated code: a large-scale comparison of large language models. https://doi.org/10.1007/s10664-024-10590-1 (2025)
23. U.S. Department of Defense: DoD Risk Management Framework (RMF). https://public.cyber.mil/rmf
24. 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)
25. U.S. General Services Administration: FedRAMP (Federal Risk and Authorization Management Program). https://www.fedramp.gov
26. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. pp. 6000–6010. NIPS'17 (2017)
27. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. In: Proceedings of the 36th International Conference on Neural Information Processing Systems. NIPS '22, Curran Associates Inc., Red Hook, NY, USA (2022)
28. x1xhlol: BOLT IDE system prompt (llama 4, december 2024). https://raw.githubusercontent.com/x1xhlol/system-prompts-and-models-of-ai-tools/

`refs/heads/main/Open%20Source%20prompts/Bolt/Prompt.txt` (2024)
29. Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T.L., Cao, Y., Narasimhan, K.: Tree of thoughts: deliberate problem solving with large language models. In: Proceedings of the 37th International Conference on Neural Information Processing Systems. NIPS '23, Curran Associates Inc., Red Hook, NY, USA (2023)