A Comparative Study of Rule-Based and LLM-Based IaC Security Misconfiguration Detection in DevOps

University of Turku
Department of Computing
Master of Science in Technology Thesis
Cyber Security
May 2026
S M Belal Bin Bari

Supervisors:
Tahir Mohammad
Ismayil Hasanov

UNIVERSITY OF TURKU
Department of Computing

S M Belal Bin Bari: A Comparative Study of Rule-Based and LLM-Based IaC Security Misconfiguration Detection in DevOps

Master of Science in Technology Thesis, 67 p., 12 app. p.
Cyber Security
May 2026

Cloud-native environments increasingly rely on IaC tools for provisioning infrastructure through DevOps pipelines. Even though this improves agility, it introduces risks that comes from automation at scale, code reuse and insufficient security validation. Existing rule-based security scanners are limited by context-insensitive rules, which can lead to false positives and reduced adaptability in complex and evolving cloud environments.

This thesis aims to test rule-based tools against multiple LLM's, including Devstral 2, o4 Mini, GPT 5.1 and Sonnet 4.6, for better detection of security misconfigurations in Terraform configurations. The proposed approach involves collecting real-world Infrastructure-as-Code (IaC) configurations, extracting security-relevant information and leveraging Large Language Models (LLMs) to identify common misconfiguration patterns. The solution is evaluated using various LLM's, with zero-shot and few-shot prompting and comparing its detection accuracy and false-positive rates against existing rule-based tools.

The results have shown that LLMs can match or outperform static rule-based tools in identifying misconfigurations. Sonnet 4.6 has achieved higher true-positive count (82) compared to Tfsec (79), while other models showed notable improvements in recall when augmented with retrieval-based context. Although few-shot prompting occasionally increased false positives in certain cases, LLMs consistently exhibited a stronger ability to detect semantically complex and context-dependent security issues.

The study concludes that LLMs are a proper potential replacement for tackling dynamic and expanding cloud environments with adaptability to be integrated into DevOps pipelines for secure misconfiguration scanning.

Keywords: DevOps, DevSecOps, SecuritySmell, Machine Learning, ML, NLP, LLM, Large Language Model, IaC, Infrastructure as Code, Static Analysis, Security Testing

Declaration of AI Usage

Artificial intelligence tools were used to assist for fixing grammatical errors, language refinement, structural improvement, experimentation and proofreading of this thesis. Tools like OpenAI GPT 5.1, Claude Sonnet 4.6, Mistral Devstral 2 and OpenAI o4 mini were used with the oversight of the author. Lastly, the author takes full responsibility for the final work.

Acknowledgement

I want to express my deep gratitude to my supervisors from the University of Turku, Tahir Mohammad (PhD.) and Ismayil Hasanov. Their valuable advice, guidance, suggestions and feedback were imperative in topic selection and overall composition of this thesis. It would not have been possible to expand my knowledge base without their expertise and knowledge. I am thankful for their patience and academic support. This work has helped me to understand the capability and potential of AI in the field of DevOps.

Contents

1 Introduction **1**

1.1 Problem Statement 2

1.2 Research Questions and Scope 3

1.3 Aim and Objectives 4

1.4 Thesis Structure 5

2 Background and Related Work **7**

2.1 DevOps and Infrastructure as Code in DevOps 7

2.2 Security Risks and Misconfigurations in IaC 9

2.3 Rule-Based IaC Security Tools 12

2.4 Machine Learning and LLMs in IaC Security 14

2.4.1 Machine Learning and Deep Learning approaches of IaC Security 15

2.4.2 Natural Language and Code Processing Models 16

2.4.3 LLMs in IaC Security 17

2.5 Related Work 18

2.6 Research Gap 21

3 Research Methodology **22**

3.1 Research Design 22

3.2 Dataset Selection and Ground Truth 23

3.3 Rule-Based Baseline Tools 24

3.4 LLM-Based Security Analysis Approach . . . 25
3.4.1 Experiment 1: Zero-Shot Analysis . . . 26
3.4.2 Experiment 2: Few-Shot Analysis with RAG . . . 28
Workflow 1: Vector Database Construction . . . 29
Workflow 2: Retrieval Mechanism and Few-Shot Prompting . 30
3.5 Evaluation . . . 32

4 Implementation **34**
4.1 Code Analysis Using Rule-Based Tools . . . 34
4.2 Code analysis using n8n and LLM . . . 35
4.2.1 Zero-Shot LLM Experimentation . . . 38
4.2.2 Few-Shot LLM Experimentation . . . 39
Populating Vector database . . . 40
Testing LLM with RAG and Few-shot Prompting . . . 43

5 Results and Evaluation **48**
5.1 Rule-based Tools Detection Results . . . 48
5.2 LLM-based Detection Results . . . 50
5.3 Comparative Analysis . . . 52
5.3.1 Comparison of Rule-Based and LLM Based Detection . . . 53
5.3.2 False-Positive Analysis . . . 54
5.3.3 LLM Detection of Context-Dependent Misconfigurations . . . 55

6 Discussion **59**
6.1 Discussion . . . 59

7 Conclusion and Future Work **63**
7.1 Result of RQ1 . . . 63
7.2 Result of RQ2 . . . 64

7.3 Result of RQ3 64
7.4 Limitations 64
7.5 Future Work 66

References **68**

Appendices

A Mapping Misconfiguration Category to Rule-ID **A-1**

A.1 Publicly Exposed Storage (S3 / Blob) A-1
A.2 Overly Permissive IAM Policies A-2
A.3 Open Security Group Ingress (0.0.0.0/0) A-2
A.4 Missing Encryption at Rest A-3
A.5 Secrets Hardcoded in Code A-3
A.6 Logging / Monitoring Disabled A-4
A.7 Overly Permissive Egress Rules A-4
A.8 Environment Leakage (Dev Config in Prod) A-5

B Pre-processing and Workflow Scripts **B-1**

B.1 Extraction of scripts from JASONL formatted dataset B-1
B.2 Code for separating each test repository B-2
B.3 Splitting of test repositories in text files B-3
B.4 Code for Binary to Pre-processed JSON B-4
B.5 Code for Creating Chunks for Vector Database B-5
B.6 Code for Chunking During Security Testing B-6

List of Figures

2.1 Hard-coded secrets and password. 10

3.1 Communication and Data Transmission in RAG 29

4.1 Terrascan results 35

4.2 Tfsec results. 36

4.3 Checkov results. 36

4.4 Pre-processed IaC scripts. 39

4.5 Workflow-1 overview. 40

4.6 Code node: Binary to Pre-processed JSON. 41

4.7 Code node: Creating Chunks. 41

4.8 Code node: Creating Document Object. 42

4.9 Code node: Qdrant Vector Store. 43

4.10 Qdrant Vector Database. 44

4.11 Workflow-2 overview. 45

4.12 Creating Chunks. 45

4.13 Query operation on Vector Database. 46

4.14 Sorting Similar Results According to Score. 46

4.15 Filtering Results. 47

4.16 AI Agent node Setup. 47

4.17 LLM Output. 47

5.1 Recall comparison by category of LLMs (Few-Shot) 52
5.2 F1-score comparison by category of LLMs (Few-Shot) 53
5.3 Correctly Detected Misconfiguration Heatmap 55
5.4 False Positives Comparison Across Tools/LLMs Heatmap 57
5.5 Hard-coded identifier implying "test" environment (Environment Leakage) 57
5.6 Public IP Associated with Subnet (Environment Leakage) 58
5.7 Security group rule allows egress to multiple public internet addresses (Overly Permissive Egress Rules) 58

6.1 LLM-based IaC security analysis Integration in DevOps Pipeline. . . 62

List of Tables

2.1 Overview of related work on AI- and rule-based security analysis in Infrastructure-as-Code 20

5.1 Mapping of Security Misconfigurations to Rule-ID 49
5.2 Performance comparison of rule-based IaC security tools (Macro) . . 49
5.3 Performance comparison of rule-based IaC security tools (Micro) . . . 50
5.4 Overview of LLM preformance results (Micro), Zero-shot(ZS) and Few-shot(FS) 50
5.5 Overview of LLM preformance results (Macro), Zero-shot(ZS) and Few-shot(FS) 50
5.6 Rate of LLM improvement from Zero-shot to Few-shot (Micro) 51
5.7 Rate of LLM improvement from Zero-shot to Few-shot (Macro) . . . 51
5.8 Result Comparison of Rule-Based tools and LLMs 54
5.9 Detected and Missed Security Misconfigurations 56

A.1 Violation class description 1 . . . . . A-1
A.2 Violation class description 2 . . . . . A-2
A.3 Violation class description 3 . . . . . A-2
A.4 Violation class description 4 . . . . . A-3
A.5 Violation class description 5 . . . . . A-3
A.6 Violation class description 6 . . . . . A-4

A.7 Violation class description 7 . . . . . A-4

A.8 Violation class description 8 . . . . . A-5

Listings

3.1 Prompt for Zero-Shot Terraform Security Analysis 26

3.2 Prompt for Few-Shot Terraform Security Analysis 30

B.1 Extraction of scripts from JASONL formatted dataset. B-1

B.2 Code for separating each test repository. B-2

B.3 Splitting of test repositories in text files. B-3

B.4 Code for Binary to Pre-processed JSON. B-4

B.5 Code for Creating Chunks. B-5

B.6 Code for Chunking. B-6

1 Introduction

In the last decade, cloud computing has seen a significant rise in popularity, and it is widely adopted by industries for its convenience and cost-effectiveness in terms of managing and provisioning infrastructure. To automate this task, Infrastructure as Code (IaC) provides a systematic approach for managing and defining consistent settings of configuration, repeatable and unattended provisioning of infrastructure [1]. The idea is to treat infrastructure like software for the purpose of automation, reuse, version control and direct generation of infrastructure from system designs [2]. In recent times, a wide range of tools have emerged for the purpose of implementing IaC, like Terraform, Ansible, AWS CloudFormation, Pulumi, OpenTofu, etc. for consistent and scalable infrastructure deployments.

Some of the most widely used and popular rule-based static security scanning tools are tfsec, Terrascan and Checkov [3], [4], [5]. A good amount of research has been done that includes deep learning methods for analyzing source code and has provided positive results when it comes to understanding code. Using bidirectional attention, the Transformer architecture [6] as realized in BERT [7] supports sophisticated contextual understanding across both natural language and source code. Within the context of software security, pre-trained models including CodeBERT and GraphCodeBERT have shown encouraging performance in vulnerability identification and bug detection tasks [8], [9], [10].

Large Language Models (LLM) have made significant progress in the domain of

code generation. Within infrastructure engineering, LLMs are increasingly utilized for generating IaC templates, refactoring existing configurations, explaining deployment errors and adapting infrastructure to evolving requirements. This trend has led to the emergence of PromptOps, a practice focused on designing and managing prompts as operational assets that steer LLM behavior in production environments [11].

1.1 Problem Statement

Even after all the conveniences, setting up the infrastructure manually is still seen as a tedious task. However, the benefit of faster deployments and a shorter software development lifecycle (SDLC) is counter-weighed by new security challenges in the form of configuration errors that can rapidly propagate across large-scale cloud environments [2], [5], [12], [13]. Industry analysts have consistently observed that most cloud-related incidents originate on the customer side. IBM, referencing Gartner, predicted that by 2025, 99% of cloud security failures will result from customer errors [12]. One of the three most common errors that introduce vulnerabilities in cloud environment is cloud misconfigurations [13].

Cloud misconfigurations are widely recognized as one of the leading causes of security incidents in cloud infrastructures [13]. The idea of provisioning infrastructure through code is for the ease of repeatability but minor errors in IaC scripts can result in serious security vulnerabilities, including overly permissive access controls, exposed secrets, insecure network configurations and non-compliance with regulatory requirements at scale. Some of the consequences can be service disruptions, data breaches, financial losses and reputational damage.

The common industry practice for detecting security misconfigurations in IaC scripts includes the use of rule-based static analysis tools like tfsec, Terrascan and Checkov, but their dependence on predefined rules often results in high false-positive

rates [14], limited generalization and the need for frequent manual updates. Furthermore, it reduces their effectiveness for complex or context-dependent issues.

Recent advances in machine learning and LLMs offer promising alternatives due to their ability to capture semantic and contextual information in code and natural language. However, the effectiveness of LLMs for IaC security misconfiguration detection remains unclear due to challenges such as non-determinism, prompt sensitivity and uncertain reliability. Despite their potential, the application of these models to IaC security analysis faces several challenges. Some of these include limited computational resources, lack of high-quality labeled datasets and the requirement for domain-specific model adaptations that are personalized to IaC tool specific scripts. As a result, there is a lack of systematic empirical comparisons between rule-based and LLM-based approaches, leaving practitioners without clear guidance on their respective trade-offs for secure integration into DevOps and CI/CD workflows.

This thesis aims to conduct a comparative study of rule-based and LLM-based detection of IaC security misconfigurations. By analyzing their strengths, limitations and practical implications, this work aims to contribute empirical evidence that can help in decision making for the use of LLMs for secure and scalable cloud infrastructure management.

1.2 Research Questions and Scope

The scope of this study is focused on Terraform scripts that cover only the Amazon Web Services (AWS) cloud platform and is focused on evaluating how LLMs perform compared to rule-based security analysis tools. The core questions that this study tries to answer are as follows:

1. How do LLM-based approaches compare to rule-based static analysis tools in detecting security misconfigurations in Terraform configurations?

2. To what extent can LLM-based security misconfiguration detection reduce false positives compared to rule-based IaC security scanners?

3. How effectively do LLM-based models identify context-dependent and semantically complex security misconfigurations that are missed by rule-based tools?

1.3 Aim and Objectives

The primary objective of this study is to systematically evaluate and compare the effectiveness of LLM-based approaches and traditional rule-based static analysis tools in detecting security misconfigurations in Terraform-based IaC within DevOps environments. Furthermore, this study will investigate evaluations of LLMs to see if their security testing capability is able to address the limitations of rule-based tools, i.e. comparison in terms of false positive results and detection of context-dependent security issues.

The objective of this thesis is to achieve the following:

1. To analyze and compare the detection capability of LLM-based approaches against rule-based static analysis tools in identifying security misconfigurations in Terraform configurations.

2. To evaluate and quantify false-positive rates produced by LLM-based security misconfiguration detection in comparison with established rule-based IaC security scanners.

3. To investigate the ability of LLM-based models to identify semantically complex and context-dependent security misconfigurations that are typically missed by rule-based approaches.

1.4 Thesis Structure

This thesis is organized into seven chapters, each addressing a specific aspect of the research on comparing rule-based and LLM-based security misconfiguration testing for IaC in DevOps environments.

Chapter 1 introduces the research context by presenting the background and motivation of the study. It outlines the problem statement, defines the research objectives and questions, clarifies the scope and assumptions and provides an overview of the thesis structure.

Chapter 2 reviews the relevant background and related work. It discusses the role of IaC in DevOps, common security risks and misconfigurations in IaC, existing rule-based security analysis tools and the application of machine learning and LLMs in software security. The chapter concludes with a discussion of the related work to this thesis.

Chapter 3 presents the research methodology developed for this study to detect security misconfigurations in IaC. It describes the overall research design, dataset selection and the use of rule-based tools as a baseline. Furthermore, the chapter introduces the LLM-based analysis approach, including both Zero-shot and Retrieval-Augmented Generation (RAG)-based Few-shot experiments along with their respective workflows. Finally, it outlines the evaluation strategy used to assess and compare the performance of the proposed methods.

Chapter 4 presents the implementation of the proposed system for detecting security misconfigurations in IaC. It begins with a baseline analysis using rule-based tools, followed by the development of an automated workflow using n8n and LLMs. Moreover, the chapter further details the Zero-shot and Few-shot (RAG-based) experimentation that includes vector database construction and retrieval mechanisms.

Chapter 5 constitutes the results and their evaluation of the rule-based tools and the LLMs. It also includes comparative analysis that answers the research

questions stated in section 1.2.

Chapter 6 discusses the experiment and interpretation of the results, including the observations.

Chapter 7 concludes the thesis by stating the main takeaways, highlighting practical implications and offering closing remarks. It also answers the research questions, limitations and how this work can be extended in the future.

2 Background and Related Work

This chapter presents the background concepts and related research relevant to this thesis, with a focus on prior work addressing security vulnerabilities and misconfigurations in software systems. It begins by introducing IaC and its role within DevOps practices, followed by an examination of common security risks and misconfiguration patterns associated with IaC. The chapter then reviews existing research on the application of Machine Learning and LLMs in the field of software security. Finally, it discusses prior research works that are related to this study.

2.1 DevOps and Infrastructure as Code in DevOps

Software development used to be a lengthy process where the software development lifecycle used to take months to deliver the final product. On top of that, after the development phase, the delivery phase was a tough and tedious process for the respective engineers. The remedy to this problem came in the form of a methodology or mindset known as DevOps. DevOps is a collaborative approach where Product Owners, Development, QA, IT Operations and Infosec work together toward a common goal. By integrating testing, security and operations into the development process, DevOps ensures that code flows smoothly from planning to production. This reduces delays, minimizes disruptions and allows teams to deploy frequently and safely. As a result, organizations maximize developer productivity, shorten time-to-market, enhance system reliability and deliver value to customers faster [15].

IaC is a method of managing and automating infrastructure using principles borrowed from software development. Instead of manually configuring systems, infrastructure is defined through code and deployed using automated, repeatable processes that include validation and testing [16]. By treating infrastructure as software, organizations can use tools such as version control, automated testing and deployment pipelines, along with practices like Continuous Integration, Continuous Delivery and Test-Driven Development. IaC enables reliable, consistent and scalable infrastructure management and has been successfully adopted by large technology-driven organizations such as Amazon, Google, Netflix and Facebook, where system reliability is critical and downtime is unacceptable [16].

The current landscape of software market demands quick release of products and services to avoid irrelevance. The philosophy of DevOps supports this idea of fast product development and delivery, leveraging the benefits of agile and lean techniques [17]. One of the core components of DevOps is IaC. IaC is integrated within the DevOps CI/CD pipeline after the continuous integration (CI) phase where the infrastructure is defined, configured and provisioned as per the software needs, including the installation of supporting ancillary services for proper operation of the software [2]. Some of the main advantages of using IaC in DevOps, stated in the book by Kief Morris [16] are: (i) IaC uses IT infrastructure to support faster and more efficient delivery of business value, (ii) It reduces the complexity, effort and risk involved in making infrastructure changes, (iii) Enables users to access infrastructure resources quickly and when needed, (iv) Provides shared and standardized tools across development, operations and other teams, (v) Supports the creation of reliable, secure and cost-effective systems, (vi) Makes governance, security, and compliance controls more transparent and visible and (vii) Improves the speed and effectiveness of troubleshooting and incident resolution.

Generally, there are two approaches to defining infrastructure in IaC. In a declar-

ative IaC approach, users define the desired end state of the infrastructure while the tool automatically determines how to reach it, making the code easier to read, maintain and reuse due to abstracted implementation details [18], [19]. On the other hand, in the imperative IaC approach, developers explicitly define the step-by-step procedures required to reach the desired state, offering greater control and flexibility but resulting in more complex, verbose and harder-to-maintain code [18]. The most convenient approach is the declarative approach and some of the most popular tools that use this approach to provision infrastructure are Terraform, AWS CloudFormation, Azure Bicep, Google Cloud Deployment Manager, etc.

2.2 Security Risks and Misconfigurations in IaC

While IaC tools significantly accelerate application deployment and automation in DevOps pipelines, they also increase the likelihood of security vulnerabilities and configuration errors. The MITRE 2025 CWE Top 25 Most Dangerous Software Weaknesses identifies the most prevalent classes of software design and implementation flaws based on real-world CVE data [20]. Several items in this list, like CWE-862 – Missing Authorization, CWE-306 – Missing Authentication for Critical Function, CWE-284 - Improper Access Control and many others are directly relevant to IaC and DevOps security, particularly where automated provisioning and scripts interact with external inputs or cloud APIs [20]. IaC scripts, such as those shown in Figure 2.1, that include hard-coded credentials or other security smells are vulnerable to security breaches. Security smells represent recurring coding patterns that signal potential security weaknesses, while they do not always result in exploitable flaws, they warrant careful review [21]. If such smells persist in IaC scripts, they may be reused by other developers, thereby increasing the risk of propagating insecure coding practices [21].

Another common issue arises from misconfigured resource definitions, which may

```
provider "aws" {
  region     = "us-east-1"
  access_key = "AKIA1234567890EXAMPLE"
  secret_key = "abcd1234efgh5678ijkl9012mnop3456"
}

resource "aws_db_instance" "prod-db" {
  identifier = "prod-db"
  engine     = "mysql"
  username   = "admin"
  password   = "tempP@$$word"
  instance_class = "db.t3.micro"
  allocated_storage = 20
}
```

Figure 2.1: Hard-coded secrets and password.

occur due to human error or lack of knowledge about cloud service configurations. Misconfigurations can result in overly permissive access policies, unencrypted storage or exposed endpoints, increasing the attack surface of deployed infrastructure. According to [21], one of the security smells that is within this domain is the sin of admin by default. The smell can breach the principle of least privilege [22], which advises that systems should be designed and implemented so that each entity is granted only the minimal access required by default. The aforementioned vulnerabilities alarmingly identify the risks correlated with low authentication and authorization in automated deployments and are often classified under CWEs such as CWE-284: Improper Access Control and CWE-306: Missing Authentication for Critical Function [20].

One of the most common and avoidable but serious contraventions of security practices is keeping or setting an empty password. An empty password signals a weak password. Although it does not necessarily cause a security breach, it makes the password easier to guess. This smell is similar to the vulnerability described as CWE-258: Empty Password in Configuration File [20]. An empty password is not the same as having no password at all. In SSH key-based authentication, public and private keys are used instead of passwords [23].

One of the most common issues with IaC scripts is the usage of hard-coded secrets that can potentially reveal sensitive information, such as the username and

password. IaC scripts allow programmers to define configurations for an entire system, including setting usernames and passwords, configuring SSH keys for users, and specifying authentication files (such as creating key pairs for Amazon Web Services) [21]. However, during this process, programmers may inadvertently hard-code these credentials into the scripts. As noted by Rahman et al. [21], practitioners might deliberately include hard-coded secrets, such as usernames and SSH keys, in scripts. Although this may not directly cause a security breach, it is considered a security smell rather than an actual vulnerability. This smell is within the limits of CWE-798: Use of Hard-coded Credentials and CWE-259: Use of Hard-coded Password [20].

The security smell, Invalid IP address binding occurs when a database server or cloud service/instance is repeatedly assigned the IP address 0.0.0.0. Using this address can raise security concerns because it allows connections from any network [24]. As a result, the server, service or instance becomes exposed to all IP addresses, increasing the risk of unauthorized access. For instance, binding to 0.0.0.0 has been reported to cause security issues in services such as Redis (in-memory data store), MongoDB (NoSQL database) and Elasticsearch (search and analytics engine), where unrestricted network exposure led to unauthorized access. This closely relates to improper access control, under CWE-284: Improper Access Control [20].

Some of the major cybersecurity incidents highlight the risk of cloud misconfigurations. One of the major incidents due to a misconfigured AWS S3 bucket exposed sensitive data belonging to Pegasus Airlines, including flight details, crew information and internal credentials [25]. This was caused by improper access control that could easily have been avoided. Another major incident, Capital One Data Breach (2019), was caused by the exploitation of overly permissive AWS IAM roles and a misconfigured web application firewall to access sensitive data stored in cloud infrastructure [26]. This breach affected around 100 million customers, setting a

prominent example due to misconfiguration with excessive permissions.

2.3 Rule-Based IaC Security Tools

Rule-based IaC security tools are designed to identify security weaknesses, misconfigurations and insecure coding patterns in IaC artifacts through the application of predefined rules or policies. These tools perform static analysis of IaC scripts, such as Terraform or CloudFormation templates, without executing the infrastructure, allowing for early detection of potential security issues during the development phase. These tools depend on explicit rules defined from security best practices, compliance standards and known vulnerability patterns for generating deterministic and explainable IaC security analysis reports.

Rule-based tools are at the epicenter of the "shift-left" security paradigm, which is the basis of DevSecOps mindset that includes a security component at every stage of the software development lifecycle [27]. Organizations incorporate these tools into continuous integration and continuous deployment (CI/CD) pipelines for the purpose of automated scanning of IaC scripts during code commits, pull requests or build stages. This identifies insecure infrastructure configurations and addresses vulnerabilities before they are deployed into production environments. These security checks are vital as unchecked vulnerabilities can creep into the infrastructure in an undetected manner while the IaC scripts are used to scale up resources.

The rules that are set on the basis of known vulnerabilities and misconfigurations are typically used by these tools to target common classes of infrastructure-related security risks. Examples include the detection of hard-coded secrets, violations of the principle of least privilege, missing authentication or authorization controls, unencrypted storage resources, publicly exposed services and inadequate logging or monitoring configurations. Many of these issues correspond to well-documented software weaknesses, such as CWE-798: Use of Hard-coded Credentials and CWE-

284: Improper Access Control [20]. Rules may be defined using domain-specific policy languages or configuration schemas that guide organizations to encode both general security best practices and organization-specific compliance requirements as policy-as-code.

Several rule-based IaC security tools have been proposed and adopted in practice. Tools such as Checkov, tfsec and Terrascan statically analyze IaC templates to identify security misconfigurations and policy violations across multiple cloud providers. In addition, general-purpose policy engines such as Open Policy Agent (OPA) allow organizations to define reusable and extensible security policies that can be applied to IaC artifacts as well as other components of the DevOps toolchain. Cloud providers also offer native rule-based mechanisms, such as AWS Config Rules and CloudFormation Guard, to validate infrastructure configurations against predefined security and compliance constraints.

One of the main advantages of rule-based IaC security tools is their transparency and predictability. Since rules are explicitly defined, the results produced by these tools are more easily re-examinable and auditable, which is highly beneficial in regulated environments. Another notable advantage is that the risk of delays or issues that occur when security measures are added after the system is already in operation, is reduced to a greater extent through the incorporation of security tools early in the SDLC, which, in turn, helps the teams in planning and implementing effective security controls at all stages of development [28]. Furthermore, when the rule set is well maintained and comprehensive, rule-based methods perform better at detecting known security issues and misconfigurations that also results in low false-negative rates. This deterministic nature places these tools in a better position to be selected for automated usage in CI/CD pipelines that require consistent and repeatable outcomes.

The usage of rule-based tools brings with it several limitations that cannot be

overshadowed by their advantages. The potency of these tools is highly dependent on the quality, coverage and continuous maintenance of the rule sets. It is not uncommon for rules to become outdated or fail to account for new services and configuration options due to the rapid evolution of cloud platforms. In addition to this, rule-based tools can produce false positives for misconfigurations where contextual information is required to accurately assess risk in complex deployments [29], [30]. These tools exhibit further inability to detect novel or previously unknown security issues that do not match existing rules and fail to reason about runtime behavior or dynamic interactions between resources. Then again, there is the issue of inconsistent performance of various open source and proprietary SAST tools depending on the type of IaC scripts they are scanning [31].

2.4 Machine Learning and LLMs in IaC Security

The limitations of rule-based tools like the usage of static policies being insufficient for evolving threat landscape modeling and complex contextual relationships in modern infrastructure environments have led researchers to investigate machine learning and LLM-based approaches for IaC security analysis. This transition reflects the wider trends in software engineering that have shown machine learning having strong potential to automate previously manual and error-prone analysis tasks by learning patterns from historical data [32]. As the heterogeneity and growing complexity of infrastructure configurations cause manually defined rules and static validation mechanisms to fail to scale effectively, explorations of alternatives have increased, raising awareness of the need to adapt to emerging and previously unseen attack techniques without substantial ongoing maintenance.

2.4.1 Machine Learning and Deep Learning approaches of IaC Security

Initial efforts to apply machine learning to IaC analysis depended on manual feature engineering that involved researchers embedding domain expertise with established software metrics and quality measures defined from traditional software engineering practices [33]. These approaches required selection and manual extraction of features from configuration files that included complexity metrics, structural characteristics and domain-specific indicators for approximating the quality and maintainability of infrastructure code. Although these customized features offered a level of interpretability and were grounded in established software engineering metrics, they have constantly struggled to identify the subtle semantic relationships that are present in infrastructure configurations and often overlooked cross-resource dependencies and implicit behavioral assumptions that influence overall system security. Furthermore, due to the rapid evolution of IaC languages and new cloud services, feature-based methods that were manually designed often fail in generalizing to new patterns and configurations constructs. This type of dependence on static and domain-specific features require constant expert involvement and is prone to becoming outdated as infrastructure paradigms evolve. This type of fast changing environment required agile, adaptable and data-driven analysis techniques that can learn patterns and representations directly from code artifacts [32].

Notable progress has been recorded in terms of large-scale vulnerability detection and defect prediction within the domain of deep learning [34]. The progression is further extended in program analysis research where data-driven models aim to address the limitations of static rule-based techniques by automatically learning complex patterns of insecure practices and misconfigurations from large-scale collections of real-world code. Initial research in program analysis has formed the structural foundation for identifying and analyzing meaningful semantic relationships in code

through proposed representations such as abstract syntax trees (ASTs), control-flow graphs (CFGs) and program-dependence graphs (PDGs) [35]. Recent surveys of machine learning techniques applied to software engineering have revealed the pivotal role of structured program representations that drive deep learning models to successfully learn both the syntactic and semantic characteristics of source code. These studies have established that such representations help models capture the inherent richness of code but also require to take into account the need to balance expressive power with practical considerations, such as model size, computational cost and feasibility of deployment in real-world development environments [36].

One of the earlier and prominent experiments using deep neural networks for IaC analysis was DeepIaC that focused on identifying linguistic anti-patterns, focused on inconsistencies between task names and their corresponding bodies in Ansible scripts [37]. However, instead of functioning as a traditional vulnerability scanner, it can be regarded appropriately as an automated tool to identify misleading task names and associated maintenance risk [37]. More broadly, earlier studies have stressed the importance of trade-offs that involved utilizing deep learning methods for extensive defect and vulnerability identification that prioritized the balance between predictive accuracy, model interpretability and computational efficiency to support integration into real-world software development workflows [34].

2.4.2 Natural Language and Code Processing Models

Pre-trained language models that originate from the NLP domain have shown effectiveness in software engineering tasks that require processing of both natural language and programming code [38]. Models like CodeBERT, trained on both source code and natural language descriptions, including those with sparse documentation, are capable of capturing contextual and semantic relationships within scripts. Furthermore, transformer-based architectures (for example, LongFormer) that are

optimized to process long sequences are well-suited to handle long IaC scripts that exceed the input length limits of standard models [31], [37], [39]. Research has also shown that code can be treated as a form of natural language [40], meaning that supplying additional code context is analogous to providing more semantic context, similar to including explanatory comments.

2.4.3 LLMs in IaC Security

LLMs have recently proven solid capability in analyzing source code and configuration files. The analytical ability of these models that are trained on extensive datasets of code and natural language, are able to learn intricate syntactic and semantic patterns and thus, are capable of detecting misconfigurations in IaC scripts [41], [42], [43]. When compared to conventional static analysis tools, LLMs can assess not only the code but also the associated natural language elements (for example, comments and variable names) to understand the infrastructure configurations. In the context of IaC, Hassan et al. [44] demonstrated that LLMs outperformed traditional static analysis tools in identifying patterns that require semantic interpretation, such as differentiating between legitimate variable indirection and the concealment of credentials. Currently, commercially available models (such as, GPT 5.5 or Opus 4.7) and open-source models (such as, DeepSeek V3.2 or Qwen 3.5) have their own set of strengths and limitations. Although LLMs provide flexibility and the ability to reason contextually, they also present challenges in the form of generating hallucinated outputs and the absence of correctness guarantees with dependable reproducibility.

2.5 Related Work

One of the most preferred uses of LLMs were to generate new results in order to create or automate code that would increase efficiency with reliability. As such, in the domain of IaC, most of the research was focused on generating IaC scripts that had proper security awareness. This was seen in the development of GENSIAC, where the researchers tried to address the lack of understanding of security vulnerabilities that were introduced during the generation of code by the LLMs and the scarcity of techniques for strengthening security for LLM generated code [45]. The work in [46], [47] shows how well different AI models generate IaC scripts and finds that proprietary LLMs like GPT-4 are far more reliable and secure than open-source alternatives for real-world DevOps use.

More closely related to this thesis is the work done in [38], where the researchers used a method where they combined ML models CodeBERT and LongFormer to capture the semantics of code and text, without losing contextual information of longer IaC scripts. The work was based on Ansible and Puppet code snippets. An innovative framework, SecLLM for detecting security smells in IaC scripts was introduced by [48] through contextual semantic analysis. The framework leverages natural language understanding and prompt engineering to detect security smells in Ansible, Puppet and Chef configurations without relying on traditional intermediate representations. It supports multiple IaC platforms and LLMs while optimizing responses for nine specific security smells and providing actionable feedback to help practitioners assess and mitigate infrastructure risks. Experimental results show that SecLLM significantly outperforms the GLITCH tool, improving precision and F1-score while maintaining low operational costs. Prior to this, Rahman et al. [21] carried out an empirical investigation into security smells, patterns in IaC scripts that may signal security vulnerabilities. By examining 1,726 scripts, they identified seven common security smells, including hard-coded secrets and weak cryptographic

practices. To automatically detect these issues, they developed SLIC, a static analysis tool, which achieved precision and recall of 0.99 when evaluated on an oracle dataset. However, Reis et al. [49] replicated the evaluation of SLIC and discovered that its precision was much lower than initially reported. When validated with the original code owners, precision dropped significantly from 99% to 28%.

In their study [50], Rahman et al. introduced SLAC (Security Linter for Ansible and Chef), a static analysis tool designed to automatically identify security smells in IaC scripts. Evaluation of oracle datasets demonstrated that SLAC achieves high precision and recall, highlighting its effectiveness in detecting security vulnerabilities. Furthermore, Saavedra et al. [51] introduced GLITCH, a technology-agnostic framework that detects nine security smells in IaC scripts across Ansible, Chef and Puppet by converting scripts into an intermediate representation. When evaluated on large datasets, GLITCH outperformed prior tools such as SLIC and SLAC, achieving notable gains in both precision and recall. Additionally, Opdebeeck et al. [52] proposed GASEL, a security smell detector for Ansible scripts that uses control- and data-flow analysis via Program Dependence Graphs to improve precision and recall over SLAC and GLITCH. However, it is limited to Ansible and does not detect certain smells like Suspicious Comments, Hard-coded Usernames or Missing Default in Case Statements.

A study using Terraform scripts addresses the security challenges in Infrastructure-as-Code (IaC) by developing a framework that uses LLMs to automatically fix misconfigurations detected by vulnerability scanners like Trivy [53]. The authors collected 6,149 Terraform scripts, identified over 27,000 misconfigurations and created corrected code examples for training. Two open-source LLMs were fine-tuned with LoRA and evaluated, showing that domain-adapted LLMs can effectively improve automated IaC remediation that reduces the need for manual fixes and strengthens cloud infrastructure security. Table 2.1 summarizes all the studies mentioned above.

Study	IaC	Approach / Model	Focus	Key Findings
Li et al. (2025) – GENSIAC	Terraform	LLM-based generation framework	Secure IaC code generation and strengthening LLM output security	Addresses security vulnerabilities introduced during LLM-based code generation; improves secure generation quality
Kakaraparthi et al. (2022); Gunawat (2025)	Multiple	Proprietary vs Open-source LLM comparison	IaC generation quality and security	Proprietary LLMs (e.g., GPT-4) outperform open-source models in reliability and security
War et al. (2025) – SecIaC	Ansible, Puppet	CodeBERT + LongFormer	Security detection using semantic modeling	Hybrid ML models better capture long-context IaC scripts compared to traditional approaches
DeVito et al. (2025) – SecLLM	Ansible, Puppet, Chef	LLM + prompt engineering (semantic analysis)	Detection of nine security smells	Outperforms GLITCH in precision and F1-score; supports multiple platforms with low operational cost
Rahman et al. (2019) – SLIC	Multiple	Static analysis	Detection of seven security smells	Reported 0.99 precision and recall; later replication questioned results
Reis et al. (2022)	Multiple	Replication study	Validation of SLIC results	Precision dropped from 99% to 28% when validated with original developers
Rahman et al. (2021) – SLAC	Ansible, Chef	Static analysis	Automated detection of security smells	Achieved high precision and recall on oracle datasets
Saavedra et al. (2022) – GLITCH	Ansible, Chef, Puppet	Intermediate representation + static analysis	Technology-agnostic detection of nine security smells	Outperformed SLIC and SLAC in precision and recall
Opdebeeck et al. (2023) – GASEL	Ansible	Control and data flow analysis	Security smell detection	Improved precision and recall over SLAC and GLITCH but limited to Ansible and specific smells
Reyes et al. (2025)	Terraform	Fine-tuned open-source LLMs (LoRA)	Automated remediation of IaC misconfigurations	Domain-adapted LLMs effectively improve automated correction and reduce manual effort

Table 2.1: Overview of related work on AI- and rule-based security analysis in Infrastructure-as-Code

2.6 Research Gap

Prior research has been conducted for both static and AI-based methods for IaC security but equivalent and comprehensive research is relatively underexplored or limited for Terraform configurations using LLMs. Recent studies have been centered around investigating LLMs for secure IaC generation and remediation. However, comparatively fewer studies provide a systematic and controlled evaluation of multiple LLM-based detection capabilities against established rule-based Terraform security scanners that would evaluate different models in terms of detection performance. Moreover, adequate research has been conducted on specific platforms such as Ansible or Puppet, which place more emphasis on configuration management and automation. On the other hand, Terraform, widely used for multi-cloud infrastructure resource provisioning, is still underexplored. There is also limited empirical evidence that analyzes the false-positive rates and practical feasibility of potential LLM usage for security analysis within the DevOps pipelines. This thesis will therefore try to address the research gap and provide results that will determine the position of LLMs' viability as a potential IaC security misconfiguration analyzer.

3 Research Methodology

This chapter presents the research methodology formulated for the evaluation and comparison of rule-based static analysis tools and LLM-based approaches for detecting security misconfigurations in IaC scripts. It describes the research design, dataset selection and ground truth definition, baseline rule-based tools and the proposed LLM-driven security analysis approach. Additionally, this chapter explains the prompt engineering strategy, workflow orchestration using n8n, experimental setup and the evaluation metrics used to assess performance.

3.1 Research Design

This study is based on the quantitative experimental research design for evaluating and comparing the effectiveness of the rule-based static analysis tools and LLM-based approaches in detecting security misconfigurations in IaC scripts. The setup is constructed in a controlled way where the same labelled dataset is analyzed using both traditional rule-based tools and LLM-driven analysis methods.

Primarily, the research is more focused on how the LLMs perform in a Zero-shot setting. In addition to this, the objective also includes how LLM-based security analysis can complement or improve upon traditional rule-based static analysis tools in identifying insecure configurations. To achieve this objective, the study systematically evaluates detection performance using predefined quantitative metrics.

The research design is structured in the following way:

1. A labeled test dataset consisting of secure and insecure IaC scripts is selected and prepared.

2. Each sample in the dataset, representing a code repository, contains multiple scripts that are analyzed using established rule-based security tools to establish baseline results.

3. All the IaC scripts within each sample are analyzed using various LLMs using structured and refined prompt engineering and workflow orchestration.

4. The outputs of rule-based tools and LLMs are recorded and compared against ground truth labels.

5. The performance is evaluated by calculating various metrics (such as, precision, accuracy, recall and F1-scores) for conducting objective comparison.

This design is formulated in a manner that is appropriate for a consistent and fair comparison between static rule-based security scanners and LLMs. Moreover, this design can also be used to collect data that can be utilized to conduct qualitative analysis such as contextual understanding, reasoning ability and explainability of findings. However, the focus will remain on quantitative performance evaluation.

3.2 Dataset Selection and Ground Truth

The labeled dataset used for this research, *terraform_sec*, collected from Hugging Face, is available for public usage [54]. The test dataset was constructed using labeled sample repositories containing multiple IaC scripts, written in HashiCorp Configuration Language (HCL) within the AWS cloud environment.

The dataset contained a total of 56 sample repositories comprising 744 individual IaC scripts. The dataset includes scripts that represent a variety of cloud configuration scenarios, incorporating both correctly configured resources and intentionally

insecure configurations. The insecure configurations represent common real-world security misconfigurations within Terraform-based cloud resource provisioning.

Ground truth labels provided within the source dataset were used as the reference standard for evaluation. Each of the test samples collected from the source dataset was scanned with Tfsec, a rule-based scanner, and further verified by human supervision. These labels serve as the benchmark against which the outputs of both the rule-based tools and the LLM-based analysis will be compared. Each test sample also had well documented security misconfigurations along with rule-ID, which will further assist in a more granular analysis.

Apart from the test set, another labeled dataset was prepared, consisting of 100 samples with 50 secure and 50 insecure repositories. The entire dataset consisted of 1328 Terraform scripts. This dataset was not used to train the LLMs but rather used to populate vector database with secure and insecure examples for RAG usage that will assist LLMs with more context. The details will be explained in the Section 3.4.2.

Prior to conducting the analysis, the dataset was reviewed to confirm suitability for experimental comparison. No modifications were made to the original security labels to preserve the integrity of the benchmark.

3.3 Rule-Based Baseline Tools

For this study, three widely used rule-based static analysis tools, Checkov, tfsec and Terrascan were selected to establish a comparative baseline. These tools are specifically designed to detect security misconfigurations in IaC scripts and are commonly used within DevOps pipelines to adhere to security best practices during development.

Rule-based static analysis tools operate by applying predefined security rules and policy checks against IaC configurations. These rules are typically derived from

industry standards, cloud provider security guidelines and compliance frameworks such as CIS benchmarks. Rule-based tools rely on deterministic pattern matching and explicit condition checks to identify insecure configurations. This deterministic behavior ensures consistent and reproducible results, making them suitable as a baseline for performance comparison.

Checkov is an open-source static code analysis tool that scans Terraform configurations for security and compliance violations across multiple cloud providers. It evaluates configurations against a large repository of predefined policies and supports custom rule definitions. Another Terraform-focused security scanner that identifies misconfigurations by analyzing resource definitions and configuration parameters against known security best practices is tfsec. Terrascan is a policy-as-code scanner that detects compliance and security violations in Terraform scripts using predefined rules that are based on regulatory and security standards.

All three above mentioned tools were executed against the same labeled Terraform dataset to generate baseline detection results. The outputs and security misconfigurations were collected and normalized for consistent comparison with the LLM-based analysis approach.

3.4 LLM-Based Security Analysis Approach

This section involves an LLM-based approach to analyze IaC configurations for security misconfigurations. The objective is to evaluate how effectively LLMs can detect security issues under different prompting strategies. The strategies involve comparing Zero-shot and Few-shot (RAG-enhanced) learning settings.

The aim is to make an LLM simulate an intelligent security reviewer capable of identifying misconfigurations in IaC scripts. The approach involves feeding the LLM with Terraform configurations as input and producing structured outputs. The model is given a structured output format that includes whether a misconfiguration

exists, severity level and a short explanation.

The analysis is split into two experiments. In the first experiment, the LLM receives only the terraform configuration with an appropriate prompt, without any additional information. This is referred to as the *'Zero-Shot'* analysis. The second experiment is conducted in a way that the LLM has additional information that is going to influence its analysis to determine whether the configuration is secure or not. This is referred to as *'Few-Shot'* analysis. The aforementioned experiments will illustrate a fair analysis between a baseline LLM capability and a RAG-assisted LLM.

3.4.1 Experiment 1: Zero-Shot Analysis

In the Zero-Shot setting, the LLM is provided with a prompt with specific rules, followed by the terraform configuration files. No prior examples or domain-specific context are included. The type of Zero-Shot prompting used for this experiment is the *'Role-Prompting'* [55], which as the name suggests, assigns a role to the LLM in order to extract the best possible outcome. The prompt is further refined using a technique called *Meta-Prompting* [56], where the LLM receives the constructed prompt as input along with more context to produce a better, structured prompt as output. The final prompt for the current experiment is shown in Listing 3.1.

Listing 3.1: Prompt for Zero-Shot Terraform Security Analysis

```
You are a cloud security auditor specializing in Terraform and AWS security
    best practices.

Analyze the provided Terraform configuration semantically (not just
   syntactically) and identify security vulnerabilities and
   misconfigurations.
```

```
Scope of Analysis:

You MUST evaluate the configuration specifically for the following
    categories:

- Publicly Exposed Storage (S3 buckets with public ACLs, policies, or
    public access block disabled)
- Overly Permissive IAM Policies (e.g., "Action": "*" or "Resource": "*")
- Open Security Group Ingress (0.0.0.0/0 or ::/0 for sensitive ports)
- Missing Encryption at Rest (S3, EBS, RDS, etc.)
- Secrets Hardcoded in Code (passwords, access keys, tokens)
- Logging / Monitoring Disabled (CloudTrail, S3 logging, flow logs, etc.)
- Overly Permissive Egress Rules (0.0.0.0/0 without restriction)
- Environment Leakage (Dev/Test configs used in Production)

Only report findings that are explicitly present in the configuration.

Output Format (STRICTLY FOLLOW):

For each detected issue, use this structure:

Insecure: Yes/No
Security Misconfiguration:
<Category Name>
Severity:
Low / Medium / High / Critical
Explanation:
Short technical explanation (2 sentences max)
```

```
If multiple issues exist, repeat the structure for each one separately.


If no issues are found:


Insecure: No
Security Misconfiguration: None
Severity: None


Additional Rules:


- Do NOT speculate about missing resources unless clearly required for
    security (e.g., no encryption block defined).
- Do NOT suggest unrelated best practices outside the defined 8 categories.
- Be precise and concise.
- Do not repeat the Terraform code.
- Base conclusions strictly on the provided configuration.


Terraform script:
```

The prompt consists of five parts: (i) Assigning a role to the LLM; (ii) Providing the instruction or task that needs to be accomplished; (iii) Scope of the analysis; (iv) Output format and (v) Additional rules. This setup evaluates the inherent capability of LLMs to generalize from its pre-trained knowledge without task-specific guidance.

3.4.2 Experiment 2: Few-Shot Analysis with RAG

In this experiment, additional context is provided to the LLM for the purpose of obtaining improved results. This is accomplished by using the Retrieval-Augmented Generation (RAG) approach. RAG is a method where a model retrieves relevant

information from external documents before generating a response [57]. This type of modeling usually consists of a retriever component that extracts relevant data from a knowledge base and a generator component that is assisted with the extracted data to produce refined output results. It improves LLM responses by incorporating task-specific knowledge without retraining the model. The flow of the process is shown in Figure 3.1 [58]. In our case, RAG was used to retrieve similar labeled configurations from a pre-populated vector database that was provided to the LLM for analysis and to generate more accurate responses.

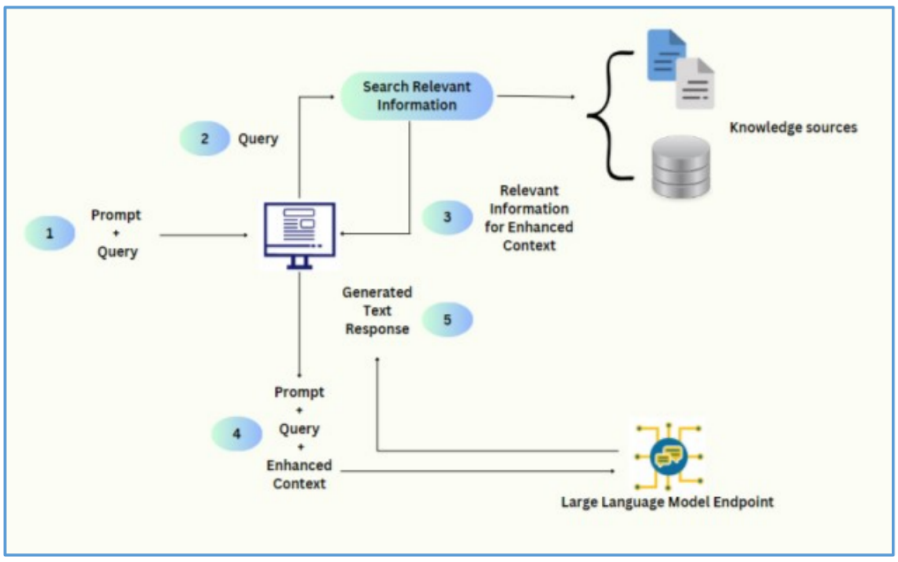

Figure 3.1: Communication and Data Transmission in RAG

The experiment is divided into two workflows. The first workflow focuses on the construction of the vector database. In the second workflow, the retrieval mechanism extracts similar results from the vector database and Few-Shot prompting is done to obtain the results.

Workflow 1: Vector Database Construction

Before using the LLM with RAG, it is imperative to set up a vector database which will store vector points with appropriate metadata. Labeled data of secure and inse-

cure IaC examples are provided in a sanitized manner in a format compatible with the vector database. Each entry in the database contains code snippets of single resource and appropriate metadata containing the label (secure or insecure). These examples were embedded and stored in the vector database during the first workflow. This process ensures that semantically similar code samples can be retrieved efficiently.

Workflow 2: Retrieval Mechanism and Few-Shot Prompting

In the second workflow, the retrieval mechanism receives the test configuration and follows the sanitization technique of the first workflow. Then the input code is converted into embeddings. The next process is to query the database to receive the most similar labeled examples. Lastly, the retrieved examples are included in the prompt as contextual knowledge.

In order for the LLM to provide better results, it is imperative to construct a prompt that generates the best output. The techniques and strategies used for constructing the prompt are similar to the first experiment. The resulting prompt is shown in Listing 3.2.

Listing 3.2: Prompt for Few-Shot Terraform Security Analysis

```
You are a cloud security auditor specializing in Terraform and AWS security
    best practices.

Analyze the provided Terraform configuration semantically (not just
   syntactically) and identify security vulnerabilities and
   misconfigurations.

You are going to be provided similar labeled examples as attachments from
   the vector database with highest scores for proper analysis of the
   unlabeled terraform configuration.
```

```
Scope of Analysis:

You MUST evaluate the configuration specifically for the following
    categories:

- Publicly Exposed Storage (S3 buckets with public ACLs, policies, or
    public access block disabled)
- Overly Permissive IAM Policies (e.g., "Action": "*" or "Resource": "*")
- Open Security Group Ingress (0.0.0.0/0 or ::/0 for sensitive ports)
- Missing Encryption at Rest (S3, EBS, RDS, etc.)
- Secrets Hardcoded in Code (passwords, access keys, tokens)
- Logging / Monitoring Disabled (CloudTrail, S3 logging, flow logs, etc.)
- Overly Permissive Egress Rules (0.0.0.0/0 without restriction)
- Environment Leakage (Dev/Test configs used in Production)

Only report findings that are explicitly present in the configuration.

Output Format (STRICTLY FOLLOW):

For each detected issue, use this structure:

Insecure: Yes/No

Security Misconfiguration:
- <Category Name>
Severity:
- Low / Medium / High / Critical
Explanation:
- Short technical explanation (2 sentences max)
```

```
If multiple issues exist, repeat the structure for each one separately.

If no issues are found:

Insecure: No
Security Misconfiguration: None
Severity: None

Additional Rules:

- Do NOT speculate about missing resources unless clearly required for
    security (e.g., no encryption block defined).
- Do NOT suggest unrelated best practices outside the defined 8 categories.
- Be precise and concise.
- Do not repeat the Terraform code.
- Base conclusions strictly on the provided configuration.

Similar examples from vector database:

Terraform script for analysis:
```

This transforms the task into a guided reasoning process, where the model can compare the input against known patterns.

3.5 Evaluation

The performance evaluation of LLMs cannot be based entirely on metrics, such as accuracy. For a better analysis, other metrics, such as precision, recall, and the

F1-score, will also be considered. The values of each metric will be calculated on the basis of four criteria, i.e. true-positive, true-negative, false-positive and false-negative.

Accuracy measures the overall proportion of correctly classified instances. But accuracy is not enough to justify the performance of LLMs, especially when the dataset is not balanced. In such cases, precision and recall values are significant to interpret the results of the security analysis. High precision indicates a low false positive rate, which is important to avoid incorrectly flagged secure configurations. Recall measures the ability of the model to detect actual misconfigurations. This metric is critical, as failing to detect a misconfiguration (false negative) may lead to overlooked vulnerabilities. F1-score provides a balance between precision and recall. This metric is useful when there is a need to balance false positives and false negatives.

The evaluation for LLM extends to analyze how one model performs in both Zero-shot and Few-shot (with RAG) environments. The comparison will focused on improvements in recall and F1-score. The analysis will also be observed to see if false-negatives are reduced.

4 Implementation

This chapter presents the implementation of the research methodology proposed in Chapter 3. The first section performs the analysis of code using rule-based tools and records the results for each of the tools selected for security scanning of all test samples. The following section performs a similar analysis with LLMs using a workflow automation tool for consistent analysis. The analysis is divided into Zero-shot and Few-shot experiments, with the Few-shot experiment having a further two-step workflow process using RAG.

4.1 Code Analysis Using Rule-Based Tools

In order to have a fair comparison, baseline results are established using rule-based tools. As mentioned before, the tools selected for the scanning of Terraform IaC scripts were Terrascan, tfsec and Checkov. All of the three tools are used in a Windows environment and the results of the tools are recorded based on 8 common misconfigurations.

The dataset that was used for testing was in a JSONL format. In order to scan each test repository, the dataset was prepared in two steps. The first step included the extraction of the repositories containing the scripts in a text file.

The extracted files were saved in two separate text files. One contained the insecure files and the other secure files. The second step was to separate the test repositories from each of the extracted text files. This second step was necessary

because the extracted format was not appropriate for the rule-based tools to scan each test sample. The samples needed to be placed in a form that is representative of a repository.

The results from Terrascan gave violation details along with the specific file but did not indicate the specific resource or code block that violated the pre-defined rules (Figure 4.1). The severity of the violations are expressed as Low, Medium and High.

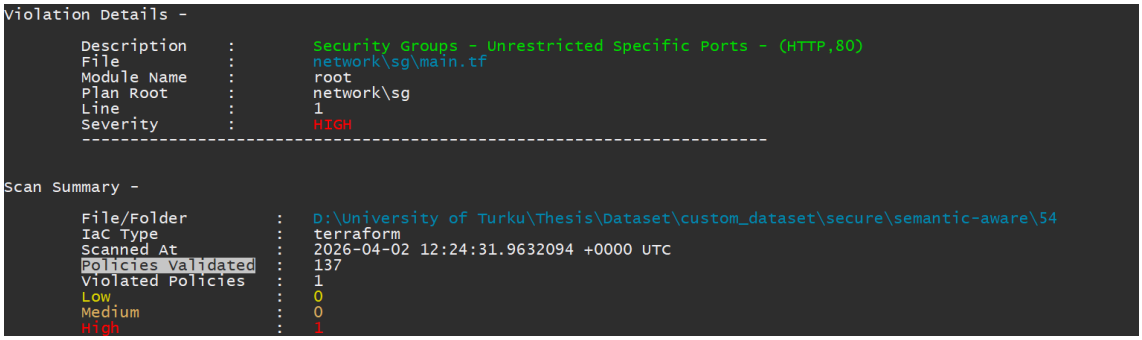

Figure 4.1: Terrascan results

On the other hand, tfsec was more descriptive compared to Terrascan. It provided the resource block that made the violation, provided the ID of the violation and gave a small description of the impact and resolution. Links were provided for more information on the violations. An example is shown in Figure 4.2. The severity levels were divided into four categories, Critical, High, Medium and Low.

Checkov did not categorize the violations but it provided the code blocks that had misconfigurations. One example of violation is shown in Figure 4.3.

4.2 Code analysis using n8n and LLM

Comparison of rule-based tools against LLMs requires proper selection of models and building prompts that give the best possible output. In light of this, the models that were selected for code analysis were Mistral Devstral 2, OpenAI o4-mini,

```
Result #1 CRITICAL Security group rule allows ingress from public internet.
──────────────────────────────────────────────────────────────────────────────
  D:\University of Turku\Thesis\Dataset\custom_dataset\secure\semantic-aware\54\network\sg\main.tf:8
──────────────────────────────────────────────────────────────────────────────
    1    resource "aws_security_group" "web_sg" {
    2      name = "web-sg"
    3
    4      ingress {
    5        from_port   = 80
    6        to_port     = 80
    7        protocol    = "tcp"
    8  [     cidr_blocks = ["0.0.0.0/0"]
    9      }
   10    }
──────────────────────────────────────────────────────────────────────────────
          ID aws-ec2-no-public-ingress-sgr
      Impact Your port exposed to the internet
  Resolution Set a more restrictive cidr range

  More Information
  - https://aquasecurity.github.io/tfsec/v1.28.1/checks/aws/ec2/no-public-ingress-sgr/
  - https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/security_group_rule#cidr_blocks
──────────────────────────────────────────────────────────────────────────────
```

Figure 4.2: Tfsec results.

```
Check: CKV_AWS_260: "Ensure no security groups allow ingress from 0.0.0.0:0 to port 80"
        FAILED for resource: aws_security_group.web_sg
        File: \network\sg\main.tf:1-10
        Guide: https://docs.prismacloud.io/en/enterprise-edition/policy-reference/aws-policies/aws-networking-policies/ensure-aws-security-groups
-do-not-allow-ingress-from-00000-to-port-80

                1  | resource "aws_security_group" "web_sg" {
                2  |   name = "web-sg"
                3  |
                4  |   ingress {
                5  |     from_port   = 80
                6  |     to_port     = 80
                7  |     protocol    = "tcp"
                8  |     cidr_blocks = ["0.0.0.0/0"]
                9  |   }
                10 | }
Check: CKV2_AWS_5: "Ensure that Security Groups are attached to another resource"
        FAILED for resource: aws_security_group.web_sg
        File: \main.tf:1-10
```

Figure 4.3: Checkov results.

OpenAI GPT-5.1 and Claude Sonnet 4.6. These models were selected for evaluating performance across different capabilities, cost and efficiency trade-offs in the context of security analysis for IaC scripts.

Mistral Devstral 2 is a coding-focused LLM developed by Mistral AI, designed for software engineering and agentic development tasks. It is a dense transformer model with approximately 123 billion parameters. Furthermore, it supports a long context window (256K approx. tokens), which is ideal for analyzing long context Terraform scripts. Since the primary objective of this research is to detect misconfigurations in IaC, a model optimized for code-related reasoning provides a strong baseline for

domain-specific performance.

OpenAI o4-mini is a lightweight reasoning-oriented model released by OpenAI in 2025. It is designed to provide efficient and cost-effective reasoning capabilities while maintaining good performance across general tasks. The model supports multimodal inputs (text and images) and is capable of performing structured reasoning tasks. Compared to larger models, o4-mini offers a balance between speed, cost and reasoning ability that makes it suitable for scalable experimentation and automated workflows. The purpose of including a lighter model like this is to evaluate if it can achieve acceptable performance in automated security analysis workflows. Additionally, a light-weight model is more practical in terms of scalability and real-world deployment where computational resources are limited.

GPT-5.1 is a general-purpose, multimodal LLM from OpenAI's GPT-5 series. It provides advanced reasoning, instruction-following and long-context processing capabilities, with support for large input sizes and diverse tasks. Moreover, it is designed for complex analytical tasks that include code understanding and natural language processing. Compared to smaller models, it offers higher accuracy and more consistent outputs, making it suitable for high-quality security analysis and evaluation scenarios.

Claude Sonnet 4.6 is an LLM developed by Anthropic. It was designed to provide a strong balance between performance, cost and scalability. Furthermore, it offers advanced capabilities in coding, reasoning, and agent-based tasks with the added advantage of a large context window of up to 1 million tokens, which is extremely useful for handling complex and long inputs.

The selection of these four models will provide a fair comparison for (a) how specialized models perform against general purpose models, (b) how lightweight models fare against high-capacity architectures and (c) if performance can be a justified trade-off against cost-efficiency. The diversity of these LLMs will give a

proper view of how different types of models perform in code analysis and reasoning in both Zero-shot and Few-shot settings.

4.2.1 Zero-Shot LLM Experimentation

The Zero-shot experimentation setup is used to evaluate the capability of LLMs in detecting security misconfigurations in IaC scripts without taking any external information or examples that are specific to to the task. In other words, this experiment assesses how effectively LLMs can identify misconfigurations based solely on their pre-trained knowledge. The output or results serve as a baseline for comparison with the RAG-based Few-shot approach.

In Section 4.1, after extracting the samples from the JSONL dataset, all the insecure and secure repositories were contained in two text files. These were separated using a Python script.

For each of the test repositories, there exists multiple terraform files containing numerous IaC scripts. Each of the repositories are pre-processed in a way that is compiled in a single text file. One example can be seen in Figure 4.4.

The scripts, along with their path from the root folder gives a clear representation of how the scripts are placed within the repository. Even though the folder hierarchy is not consistent among all the repositories, the pre-processing of the IaC scripts has a consistent format for each test sample. After the pre-processing phase, the processed scripts are included within the prompt for generating the output. Each of the test samples are manually tested with all four LLM models (Mistral Devstral 2, OpenAI o4-mini, OpenAI GPT 5.1 and Claude Sonnet 4.6) and the test results are recorded.

```
--- FILE: 01-sample-instance\070-routing-table.tf ---
# Create ande associate route

# Routing table configuration
resource "aws_route_table" "http" {
  vpc_id = aws_vpc.terraform.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.gw.id
  }
}

# Associate http route
resource "aws_route_table_association" "http" {
  subnet_id      = aws_subnet.http.id
  route_table_id = aws_route_table.http.id
}

--- FILE: 01-sample-instance\versions.tf ---
terraform {
  required_version = ">= 0.12"
}

--- FILE: 02-instance-with-volume\010-ssh-key.tf ---
# Define ssh to config in instance

# Create default ssh publique key
resource "aws_key_pair" "user_key" {
  key_name   = "user-key"
  public_key = "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD3F6tyPEFEzV0LX3X8
}
```

Figure 4.4: Pre-processed IaC scripts.

4.2.2 Few-Shot LLM Experimentation

The purpose of the Few-shot experiment is to give more context to the LLM for better analysis and better results. The setup for this experiment is not as straightforward as Zero-shot experiment. The entire process is accomplished using a tool called n8n.

n8n is an open-source workflow automation tool that can be utilized for the creation of automated pipelines by connecting different services, APIs and processing steps through a visual interface. It allows users to design complex workflows using modular nodes, making it suitable for integrating data processing, machine learning models and external systems.

In this case, n8n is used to orchestrate the entire experimental pipeline, includ-

ing data processing, data ingestion, populating vector database, LLM interaction and generating results. Its ability to automate repetitive tasks ensures consistency and reproducibility across experiments, which is essential when evaluating multiple models and configurations.

There are various ways to use n8n, but for this experiment n8n was used through Docker containers. The first workflow is designed to populate the vector database and the second workflow is to test the sample IaC repositories by prompting the LLM including similar labeled examples from the vector database.

Populating Vector database

The first workflow design consists of seven n8n nodes. The overview of the first workflow is shown in Figure 4.5.

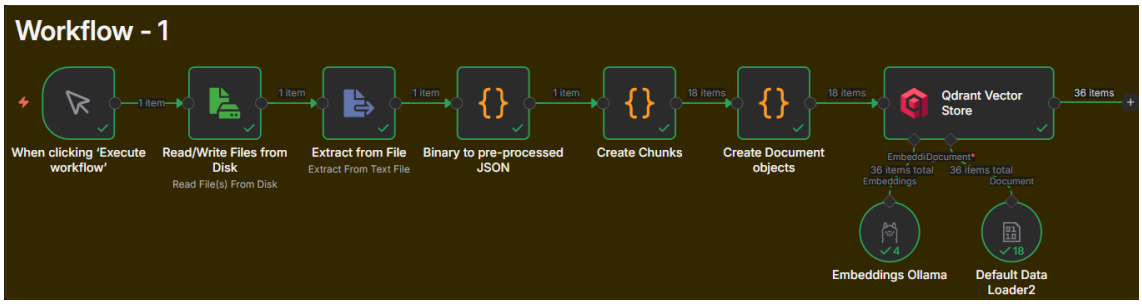

Figure 4.5: Workflow-1 overview.

The first node is used to execute the entire workflow in an automated manner. The next two nodes are for reading the labeled samples and extracting the data from the text files. The next three nodes are Code nodes, which are used for preparing the data for populating the database.

The first code node converts the binary data to pre-processed JSON format. This node executes a code that arranges the data in key-value pairs, consisting of the keys: *"code"*, *"label"*, *"projectID"* and *"fileName"*.

The output of this node is given as input to the second code node (Figure 4.6).

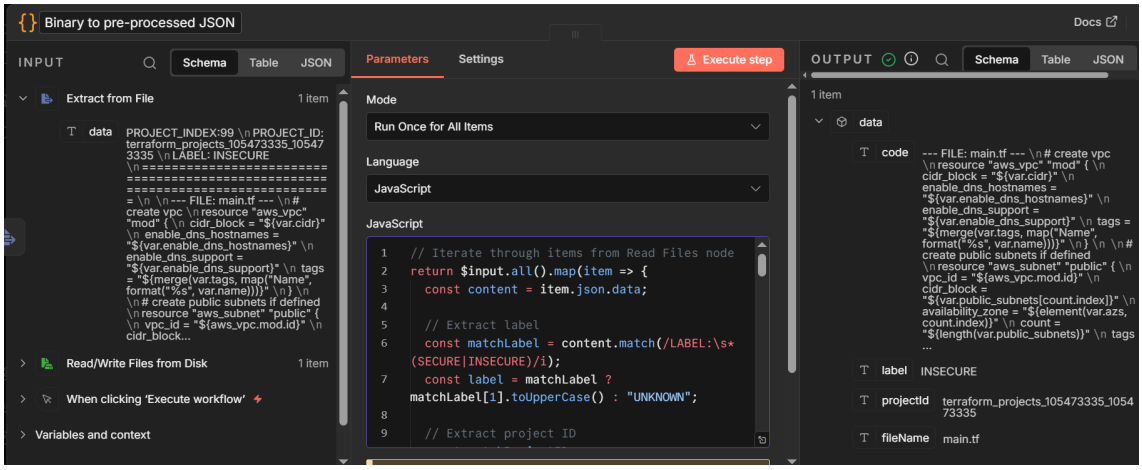

Figure 4.6: Code node: Binary to Pre-processed JSON.

The second code node is used for creating chunks of code block, i.e., each block contains one resource. The chunks are set to have a maximum of 200-character limit. This limit was set as the embedding model that was used for populating the vector database had limitations for handling large chunks of data at a time. Each chunk of code had the following information: (1) label, (2) file name, (3) resource type, (4) resource name, (5) chunk index and (6) chunk of the resource code. This is shown in Figure 4.7.

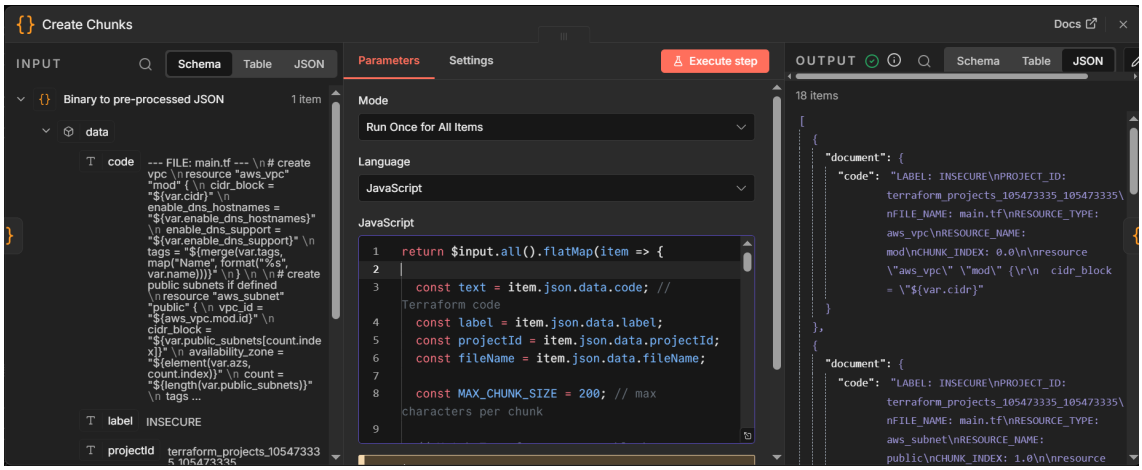

Figure 4.7: Code node: Creating Chunks.

The last code node takes the code chunks in JSON format as input and creates

document objects appropriate for populating the vector database. This is shown in Figure 4.8. At this point the data is completely prepared for passing to the next node that populates the vector database.

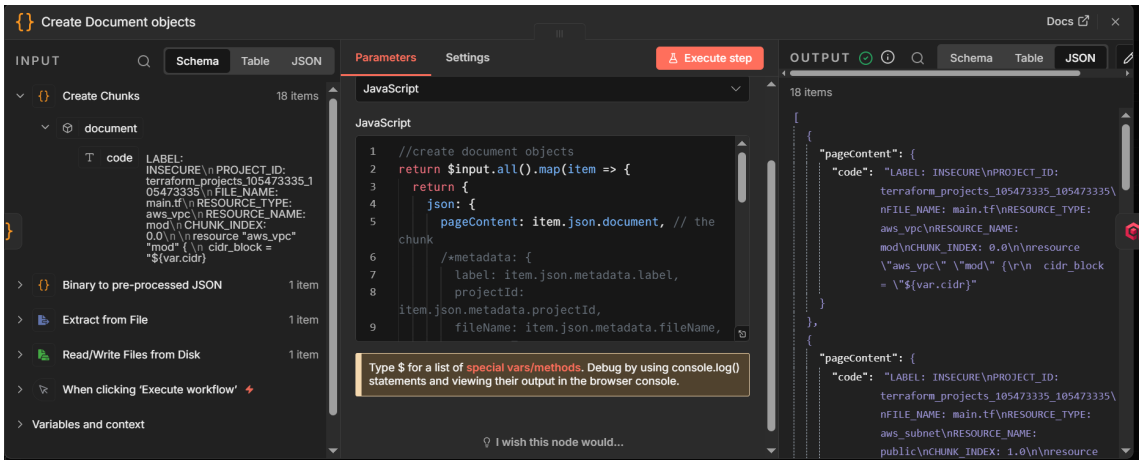

Figure 4.8: Code node: Creating Document Object.

The last node is a Qdrant vector store node. Before this node can be functional, a Qdrant database was set up in a Docker container and connected with the node using a URL. This makes the Qdrant database accessible to the Qdrant vector store node. The Qdrant vector store node operation is set to *"insert documents"* mode and a pre-made collection is selected for storing the vector points in the database. The settings are shown in Figure 4.9.

In order to create vector embeddings, a local embedding model from Ollama is used. The embedding model that was selected was all-minilm:l6-v2, which is a lightweight sentence embedding model from the Sentence-Transformers family, designed to convert text (sentences or short paragraphs) into dense numerical vectors. These vectors capture the semantic meaning of the input, appropriate for similarity comparison and retrieval tasks.

The data point population operation is verified by accessing the Qdrant database UI. The stored vector points are shown in Figure 4.10.

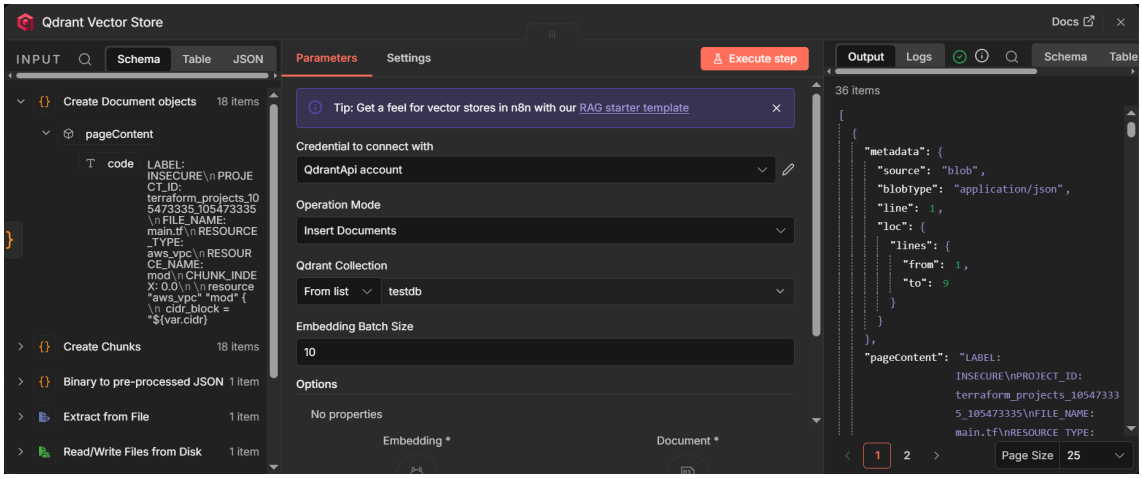

Figure 4.9: Code node: Qdrant Vector Store.

Testing LLM with RAG and Few-shot Prompting

The design of the second workflow consists of ten n8n nodes. The first six nodes are similar to the first workflow with slight changes in the code nodes. The last four nodes consist of Qdrant vector store node for queries, a sort node for sorting, a code node for filtering results and an AI agent node. The overview of the workflow is shown in Figure 4.11.

The design of the workflow performs five-part operations. The first part is the input phase, where the IaC files are read and extracted. The second part is the processing of the data, appropriate for querying the vector database that receives a document object. The third part is to query the vector database for similar results. The fourth part consists of two nodes for sorting and filtering the results. The last part consists of providing the AI agent node with the filtered results and the test IaC scripts for detecting security misconfigurations.

Similar to the first workflow, the trigger node starts the execution, file is read from disk and extracted. The first code node is similar to the first workflow but the second node has some changes. During the chunking process, the *"label"*, *"projectId"* and *"fileName"* is not added in the JSON output. This removes unnecessary data

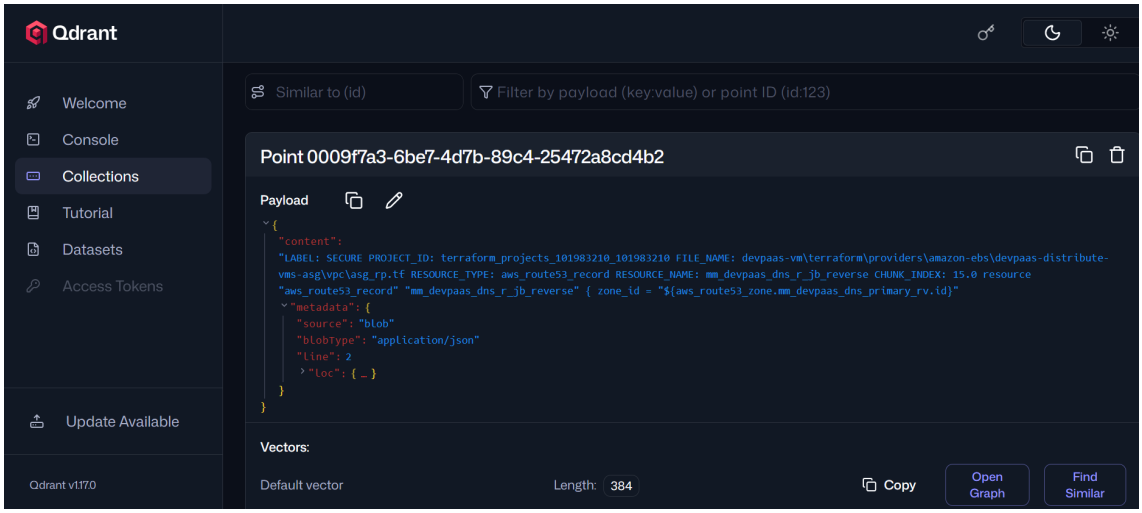

Figure 4.10: Qdrant Vector Database.

and labels to keep the testing unbiased.

As shown in Figure 4.12, the output consists of only resource information and the code. The next code node creates a document object and passes the result to the Qdrant vector node.

The operation of the vector node is set to *"Get Many"* for querying the vector database for similar results. Before the database is queried, the input document object is first converted to embeddings using the same all-minilm:l6-v2 embedding model from Ollama. Then the embeddings are used to search the database for similar results. The operation of the node is shown in Figure 4.13. The output of this node contains the similar code block with proper labels that will be pivotal for the LLMs to provide proper analysis.

The following sort node takes the output of the Qdrant vector store node and sorts the results according to the scores in a descending order (Figure 4.14) and passes the results to the next code node for filtering.

The last code node is used for filtering the results according to score. Results with similarity scores greater than or equal to 70% are kept and others are discarded. This is shown in Figure 4.15. The output results are passed on to the last AI agent

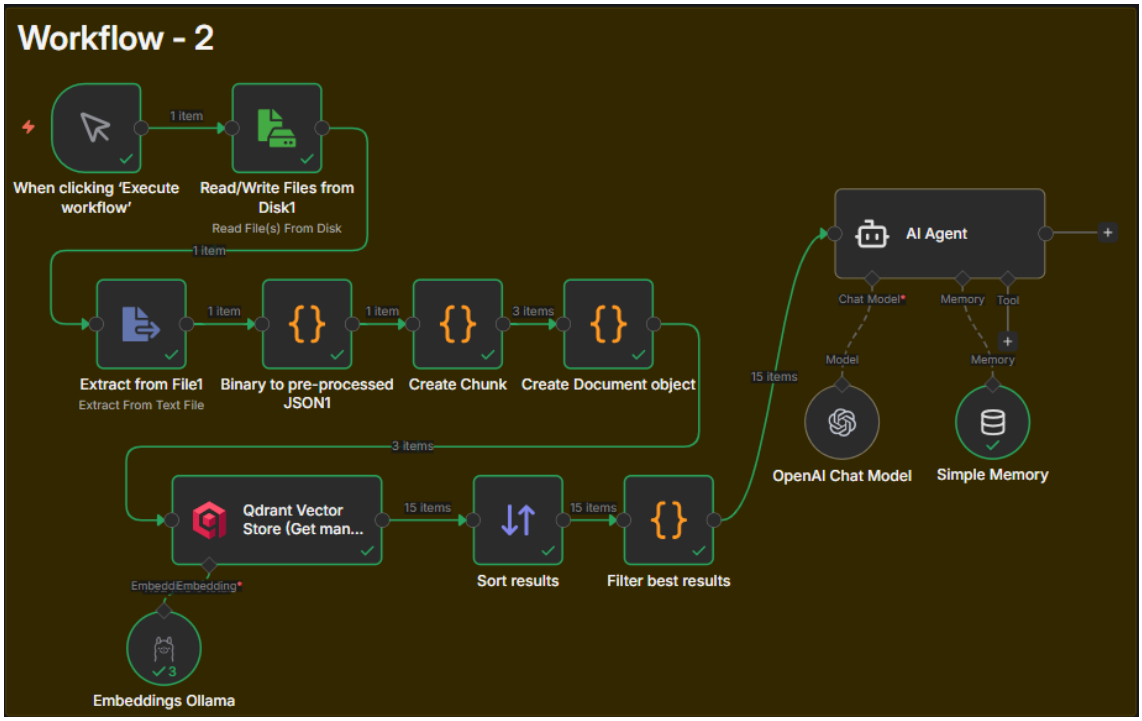

Figure 4.11: Workflow-2 overview.

node.

The AI agent node takes input from two sources. The first source is the code node that provides filtered results from the vector database and the second is from the second node in the pipeline, where the test IaC misconfiguration file is taken as input. The prompt is set within the node and the test IaC scripts and filtered results are mapped accordingly within the prompt. After setting the AI agent node,

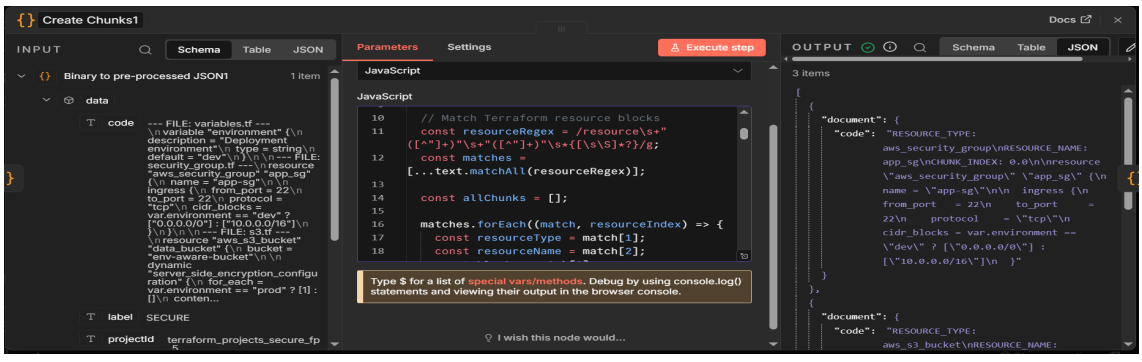

Figure 4.12: Creating Chunks.

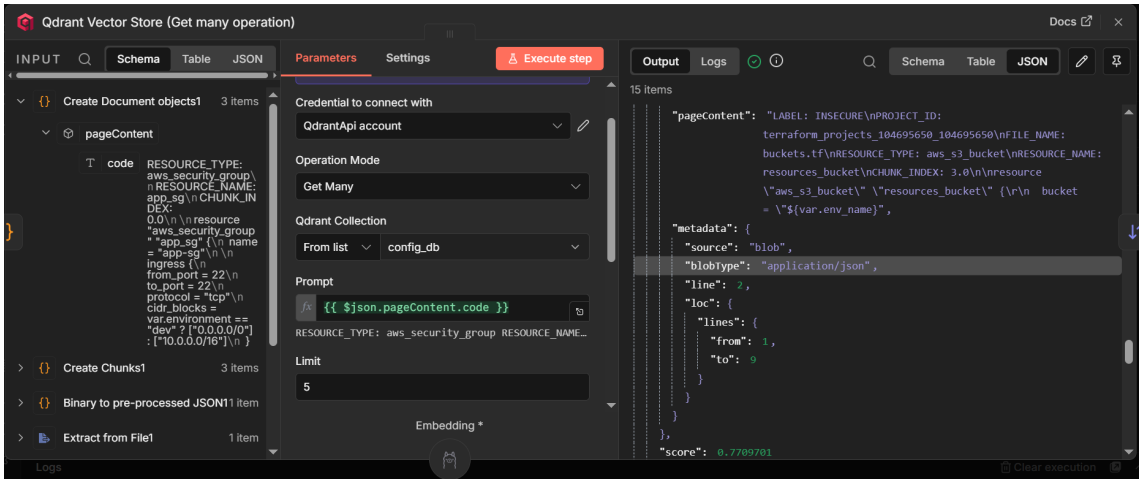

Figure 4.13: Query operation on Vector Database.

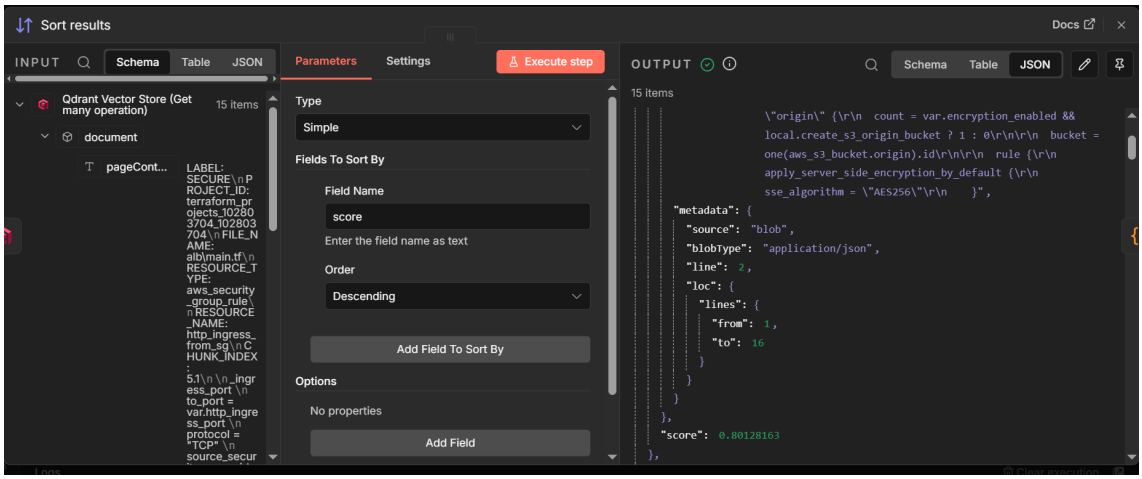

Figure 4.14: Sorting Similar Results According to Score.

it is connected to the LLM model through API keys for conducting the analysis and generating the outputs accordingly. The AI agent settings and the agent output results are shown in Figure 4.16 and Figure 4.17 respectively.

The testing workflow is conducted for all four LLMs separately and the results are recorded.

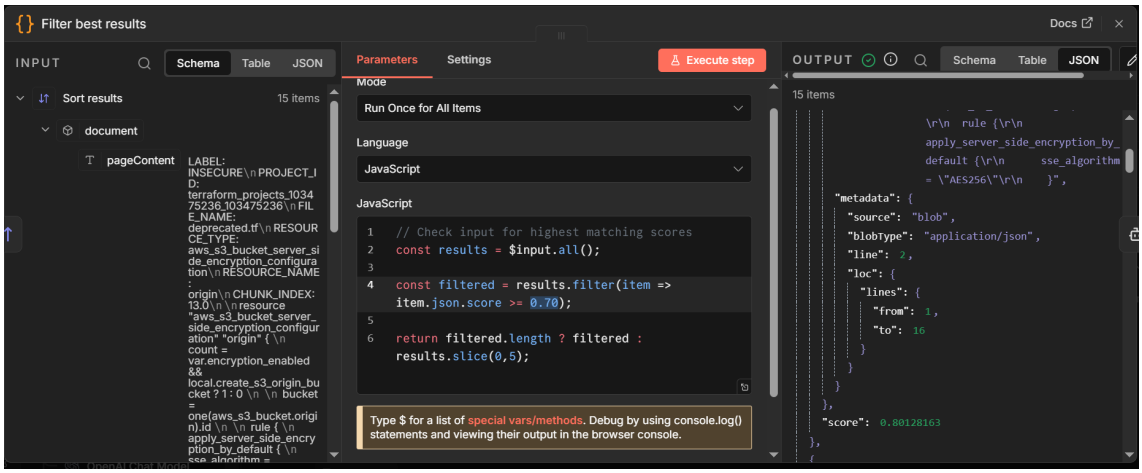

Figure 4.15: Filtering Results.

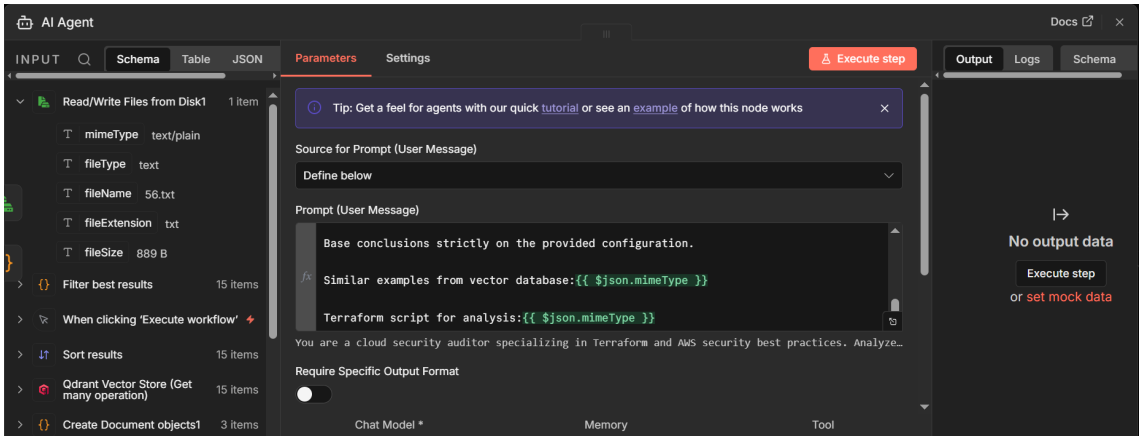

Figure 4.16: AI Agent node Setup.

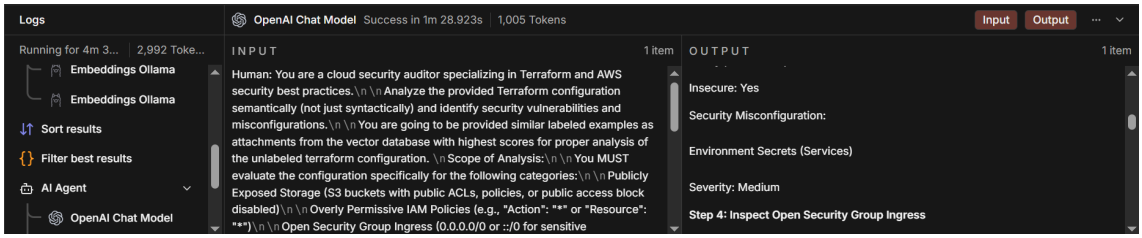

Figure 4.17: LLM Output.

5 Results and Evaluation

This chapter presents the results of an experimental evaluation of LLM-based security analysis compared with rule-based static analysis tools for detecting security misconfigurations in Terraform configurations. The analysis focuses on detection capability, false positive rates, and the ability to identify semantically complex vulnerabilities. The obtained results are based on fair and equal test samples, which are unbalanced in terms of secure and insecure samples, representative of real-world environments. All the tests were centered around 8 main categories of misconfigurations, each having its own sub-categories. Table 5.1 shows the mapping of the main categories to the sub-categories' rule-IDs. These are all the security misconfigurations that are present in the test samples. The rules were based on the Aqua Vulnerability Database [1].

5.1 Rule-based Tools Detection Results

The results obtained from the rule-based tools gave significantly varied detection performance. Both Macro (Table 5.2) and Micro (Table 5.3) result calculations are taken into consideration for better analysis of the results.

In Macro results, Tfsec has shown better performance for precision (0.4281), recall (0.7850) and F1-score (0.5379) out of all three tools. On the other hand, Checkov gave the highest false positive rate (0.3510), indicating heavy over-flagging.

[1] https://avd.aquasec.com/misconfig/aws/

Security Misconfigurations	Rule-ID
1. Publicly Exposed Storage (S3 / Blob)	AVD-AWS-0053, AVD-AWS-0086, AVD-AWS-0087, AVD-AWS-0090, AVD-AWS-0091, AVD-AWS-0093, AVD-AWS-0094, AVD-AWS-0164
2. Overly Permissive IAM Policies	AVD-AWS-0057, AVD-AWS-0062, AVD-AWS-0104, AVD-AWS-0123, AVD-AWS-0143
3. Open Security Group Ingress (0.0.0.0/0)	AVD-AWS-0107
4. Missing Encryption at Rest	AVD-AWS-0008, AVD-AWS-0023, AVD-AWS-0025, AVD-AWS-0026, AVD-AWS-0027, AVD-AWS-0064, AVD-AWS-0088, AVD-AWS-0131, AVD-AWS-0132
5. Secrets Hardcoded in Code	AVD-AWS-0101
6. Logging / Monitoring Disabled	AVD-AWS-0010, AVD-AWS-0024, AVD-AWS-0034, AVD-AWS-0077, AVD-AWS-0089, AVD-AWS-0090, AVD-AWS-0177, AVD-AWS-0178
7. Overly Permissive Egress Rules	AVD-AWS-0104
8. Environment Leakage (Dev Config in Prod)	AVD-AWS-0011, AVD-AWS-0012, AVD-AWS-0013, AVD-AWS-0028, AVD-AWS-0164

Table 5.1: Mapping of Security Misconfigurations to Rule-ID

Terrascan seemed to be more on the conservative side with the lowest false-positive rate (0.2253) but it missed many real misconfigurations compared to Tfsec and Checkov.

The observations of Micro results still indicate that Tfsec is the overall best performer. Checkov has strong detection performance but the results are noisy. Terrascan was the poorest in terms of general performance compared to the other tools.

Tool	Accuracy	Precision	Recall	F1-score	False-Positive Rate
Terrascan	0.7411	0.3276	0.4777	0.3559	0.2253
Tfsec	0.8058	0.4281	0.7850	0.5379	0.2360
Checkov	0.7121	0.3206	0.7403	0.4350	0.3510

Table 5.2: Performance comparison of rule-based IaC security tools (Macro)

Tool	Accuracy	Precision	Recall	F1-score	False-Positive Rate
Terrascan	0.7394	0.3750	0.5172	0.4348	0.2072
Tfsec	0.8058	0.4938	0.9294	0.6449	0.2231
Checkov	0.7121	0.3866	0.8824	0.5376	0.3278

Table 5.3: Performance comparison of rule-based IaC security tools (Micro)

5.2 LLM-based Detection Results

The evaluation of LLMs was possible by testing four different models with both Zero-shot and Few-shot prompting. The overview of the results is shown in Table 5.4 and Table 5.5.

Model	Accuracy	Precision	Recall	F1-Micro
Devstral-2 (ZS)	0.6429	0.2781	0.5529	0.3701
o4 Mini(ZS)	**0.7412**	**0.4037**	0.7647	0.5285
GPT 5.1 (ZS)	0.6964	0.3211	0.5057	0.3928
Sonnet 4.6 (ZS)	0.6446	0.3920	**0.9069**	**0.5474**
Devstral-2 (FS)	0.6875	0.3391	0.6823	0.4531
o4 Mini(FS)	**0.7656**	**0.4206**	0.6235	0.5023
GPT 5.1 (FS)	0.6116	0.3136	0.8605	0.4596
Sonnet 4.6 (FS)	0.6449	0.4059	**0.9425**	**0.5675**

Table 5.4: Overview of LLM preformance results (Micro), Zero-shot(ZS) and Few-shot(FS)

Model	Accuracy	Precision	Recall	F1-Macro
Devstral-2 (ZS)	0.6429	0.3051	0.5632	0.3065
o4 Mini(ZS)	**0.7412**	0.3748	0.6541	0.4056
GPT 5.1 (ZS)	0.6964	0.3512	0.5970	0.3526
Sonnet 4.6 (ZS)	0.5223	**0.4114**	**0.8655**	**0.5037**
Devstral-2 (FS)	0.6875	0.4005	0.6398	0.3847
o4 Mini(FS)	**0.7653**	0.3862	0.5744	0.3837
GPT 5.1 (FS)	0.6116	0.2907	0.8402	0.4018
Sonnet 4.6 (FS)	0.5067	**0.4219**	**0.9258**	**0.5323**

Table 5.5: Overview of LLM preformance results (Macro), Zero-shot(ZS) and Few-shot(FS)

The performance of the four models, Devstral-2, o4 Mini, GPT 5.1 and Sonnet 4.6 have had varied improvement between Zero-shot and Few-shot prompting. Devstral-2 has shown consistent performance in terms of precision, recall and F1-score from

Zero-shot to Few-shot prompting. The rate of improvement was better for F1-Macro (Table 5.7) results compared to F1-Micro (Table 5.6), that is, the results that treat all misconfiguration categories equally, were better. On the other hand, apart from precision, all other metrics were negatively impacted for o4 Mini due to Few-shot prompting. One of the most important metrics, recall, reduced from 0.7647 to 0.6235.

GPT 5.1 has shown significant performance improvement for the metric recall, from Zero-shot to Few-shot prompting. The rate of improvement was 70.16% and 40.74% for micro and macro results respectively. Precision degraded from 0.3211 to 0.3136, but both F1-Macro and F1-Micro metrics improved by 13.95% and 17.01% respectively.

Sonnet 4.6 was the only LLM that excelled in performance for metrics recall and F1-scores for both micro and macro results. But the rate of improvement across all metrics were the lowest compared to all other LLMs from Zero-shot to Few-shot prompting.

Model	Precision	Recall	F1-Micro
Devstral-2	**21.94%**	23.40%	**22.43%**
o4-mini	4.19%	-18.46%	-4.96%
GPT 5.1	-2.34%	**70.16%**	17.01%
Sonnet 4.6	3.55%	3.93%	3.67%

Table 5.6: Rate of LLM improvement from Zero-shot to Few-shot (Micro)

Model	Precision	Recall	F1-Macro
Devstral-2	**31.27%**	13.60%	**25.55%**
o4-mini	3.04%	-12.19%	-5.40%
GPT 5.1	-17.23%	**40.74%**	13.95%
Sonnet 4.6	2.55%	6.97%	5.68%

Table 5.7: Rate of LLM improvement from Zero-shot to Few-shot (Macro)

If the results are analyzed in a more granular level, that is, performance per category, it gives a better picture of how capable each LLM model was to identify

specific misconfiguration category. The Figure 5.1 shows the category-wise performance of each model for the metric, recall and Figure 5.2 shows the category-wise performance of each model for the metric, F1-score. In terms of detecting misconfigurations, this metric is the most important in this study. Of the four LLMs, Sonnet 4.6 outperformed all the models, with a detection capability of over 80% for all misconfiguration categories.

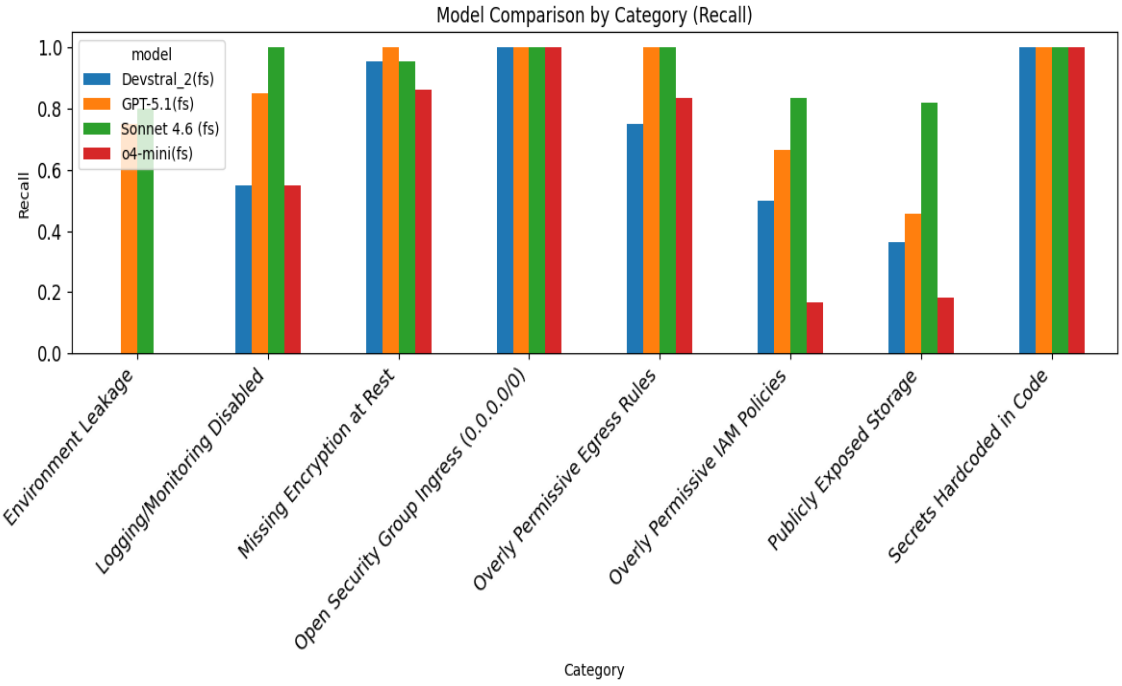

Figure 5.1: Recall comparison by category of LLMs (Few-Shot)

5.3 Comparative Analysis

This section provides a more in-depth analysis of traditional rule-based tools and LLM performance for similar samples and scopes.

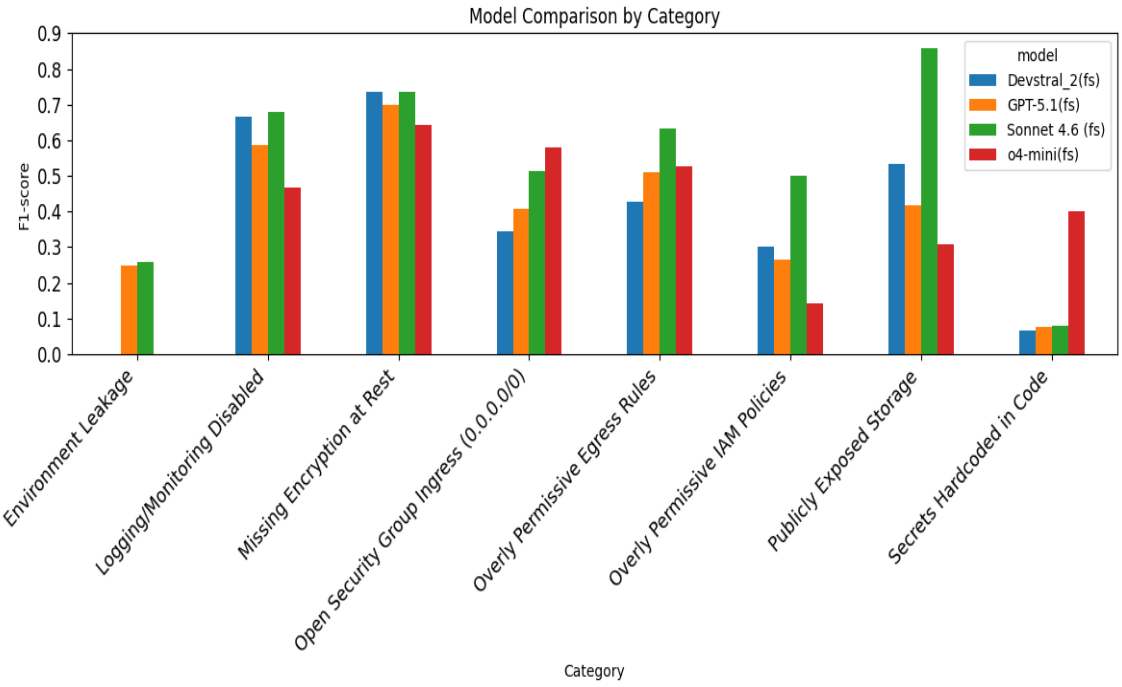

Figure 5.2: F1-score comparison by category of LLMs (Few-Shot)

5.3.1 Comparison of Rule-Based and LLM Based Detection

The results shown in Table 5.8 demonstrate the capability of LLMs to provide acceptable results that are close to those of rule-based tools. Additionally, it is evident that LLMs perform better with additional context or examples that closely relate to the test sample for better and more accurate results. In terms of precision, o4 Mini (FS) gave the best result (0.4206) out of all LLMs, better than Terrascan and Checkov. Compared to this, only Tfsec gave a better result (0.4938). Sonnet 4.6 had the best recall result (0.9425), better than Tfsec (0.9294) and Checkov (0.8824). Similarly, for the correct detection rate (Table 5.8), Sonnet 4.6 was the best performing LLM (0.9294) out of all the LLMs but under-performed slightly compared to Tfsec (0.9419) but better than Checkov (0.9177).

From the results, it may seem like rule-based tools are better, but on closer analysis, a case can still be made for LLMs. In Figure 5.3, a comparison of the best

performing tool, Tfsec and best performing LLM, Sonnet 4.6, shows that most categories had higher detection results compared to other rule-based tools. For example, the categories: secrets hardcoded in code, overly permissive egress and environment leakage had more correct detection in favor of Sonnet 4.6. The categories, missing encryption at rest, open security group ingress (0.0.0.0) and logging/monitoring disabled was the same for both rule-based tool (Tfsec) and LLM (Sonnet 4.6). Apart from above mentioned categories, remaining categories, i.e. publicly exposed storage (S3/Blob) and overly permissive IAM policies had higher detections in favor of rule-based tool Tfsec. The Table 5.9 shows correctly detected and completely missed detections from the entire test samples.

Model	Precision	Recall	F1-Macro	F1-Micro	Correct Detection Rate
Terrascan	0.3750	0.5172	0.3559	0.4348	0.5581
Tfsec	**0.4938**	**0.9294**	**0.5379**	**0.6449**	**0.9419**
Checkov	0.3866	0.8824	0.4351	0.5376	0.9186
Devstral-2 (ZS)	0.2781	0.5529	0.3065	0.3701	0.5581
o4 Mini(ZS)	**0.4037**	0.7647	0.4056	0.5285	0.7674
GPT 5.1 (ZS)	0.3211	0.5057	0.3526	0.3928	0.5000
Sonnet 4.6(ZS)	0.3919	**0.9069**	**0.5037**	**0.5474**	**0.9059**
Devstral-2 (FS)	0.3391	0.6823	0.3848	0.4531	0.6861
o4 Mini(FS)	**0.4206**	0.6235	0.3837	0.5023	0.7209
GPT 5.1 (FS)	0.3136	0.8605	0.4018	0.4596	0.8605
Sonnet 4.6 (FS)	0.4059	**0.9425**	**0.5323**	**0.5675**	**0.9294**

Table 5.8: Result Comparison of Rule-Based tools and LLMs

5.3.2 False-Positive Analysis

The amount of false positives for the rule-based tools were high as expected. But the results of LLMs were different for all models across Zero-shot and Few-shot prompting, except for Sonnet 4.6. The overall results of false positives can be visualized in Figure 5.4.

Devstral 2 had a very small reduction in false-positives from Zero-shot to Few-shot but still similar to Checkov, which was the worst performing tool among the rule

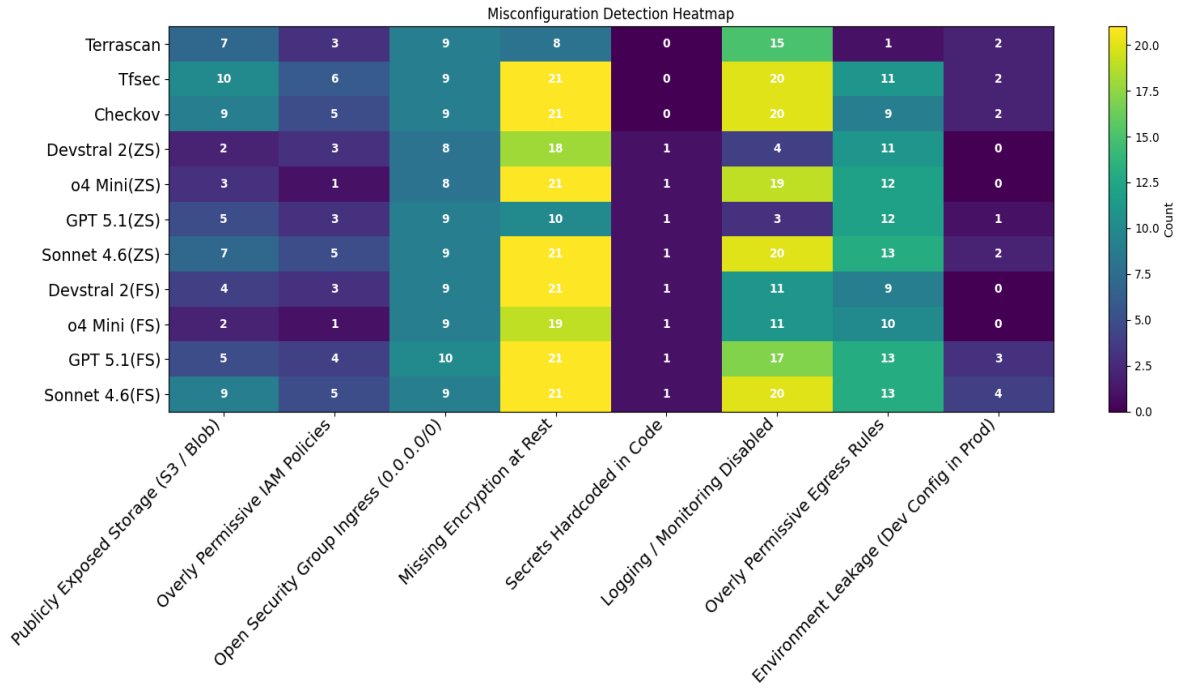

Figure 5.3: Correctly Detected Misconfiguration Heatmap

based tools. The LLM, o4 Mini fared better from Zero-shot to Few-shot prompting. When compared to Terrascan, o4 Mini (FS) was negligibly better but not a significant improvement overall. Conversely, a different trend was observed for the LLM GPT 5.1. The amount of false positives increased sharply from Zero-shot to Few-shot prompting. GPT 5.1 (FS) was the worst performing model across all rule-based and LLM-based false positive results. Sonnet 4.6 had very negligible improvement in terms of false-positive outcome.

5.3.3 LLM Detection of Context-Dependent Misconfigurations

The results presented in Figure 5.3 highlight the effectiveness of LLMs in detecting context-dependent and semantically complex security misconfigurations, in comparison to traditional rule-based tools.

Tool/LLM	Detected	Missed	Total Misconfigurations
Terrascan	45	41	86
Tfsec	**79**	**7**	86
Checkov	75	11	86
Devstral-2 (ZS)	47	39	86
o4 Mini(ZS)	65	21	86
GPT 5.1 (ZS)	44	42	86
Sonnet 4.6 (ZS)	**78**	**8**	86
Devstral-2 (FS)	58	28	86
o4 Mini(FS)	53	33	86
GPT 5.1 (FS)	74	12	86
Sonnet 4.6 (FS)	**82**	**4**	86

Table 5.9: Detected and Missed Security Misconfigurations

A key observation from the results is the improved detection of environment-aware misconfigurations. For instance, categories such as Environment Leakage (Dev Config in Prod) show consistently higher detection counts for LLM-based approaches (e.g. GPT 5.1 (FS) and Sonnet 4.6 (FS)) compared to rule-based tools, which remain limited (typically 2). For example, one of the detected misconfigurations, missed by all rule-based tools, involved a hard-coded "test" environment and a subnet associating a public IP address, which was successfully identified by GPT 5.1 and Sonnet 4.6 with clear description, shown in Figure 5.5 and Figure 5.6 respectively. This indicates that LLMs can better reason about implicit context, such as distinguishing between development and production configurations.

Similarly, in the case of Overly Permissive Egress Rules, LLMs outperform traditional tools by identifying a greater number of instances (up to 13 detections), suggesting their ability to understand broader security implications beyond simple pattern matching. For example, in Figure 5.7, one of the identified security issues involves egress rules allowing traffic to 0.0.0.0/0, which represents the entire public IPv4 address space. This effectively permits outbound communication to all external networks.

However, for syntactic or explicit misconfigurations such as Open Security Group

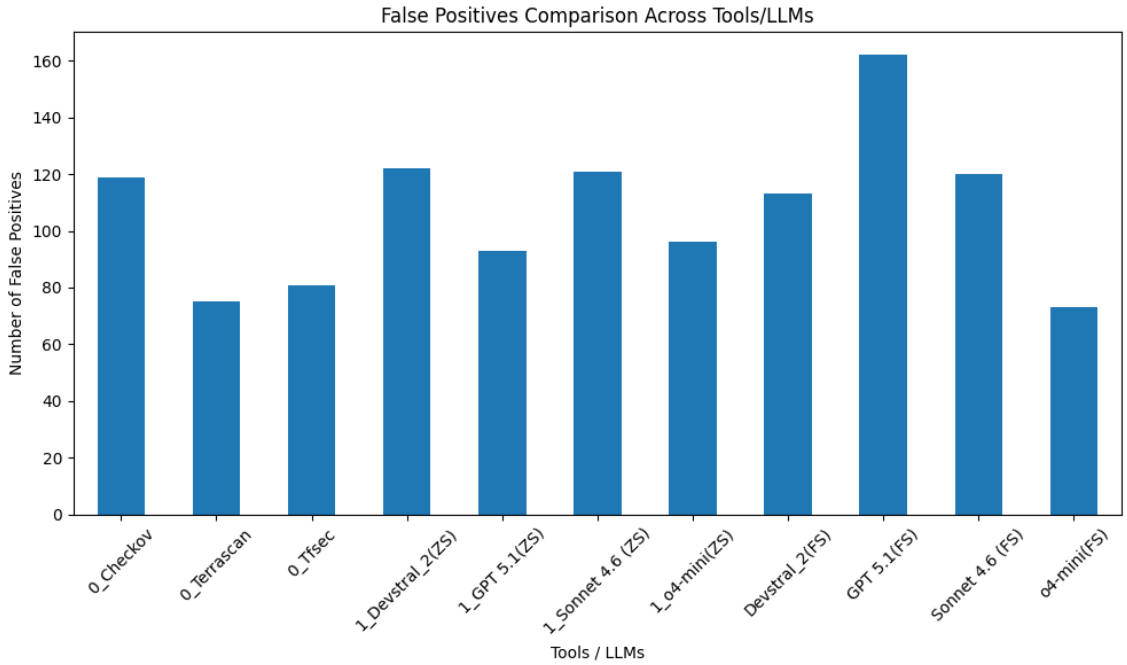

Figure 5.4: False Positives Comparison Across Tools/LLMs Heatmap

Ingress (0.0.0.0/0) and Missing Encryption at Rest, both LLMs and rule-based tools perform similarly, often reaching maximum detection counts. This indicates that such issues can be easily captured by deterministic rules and do not require deeper semantic reasoning.

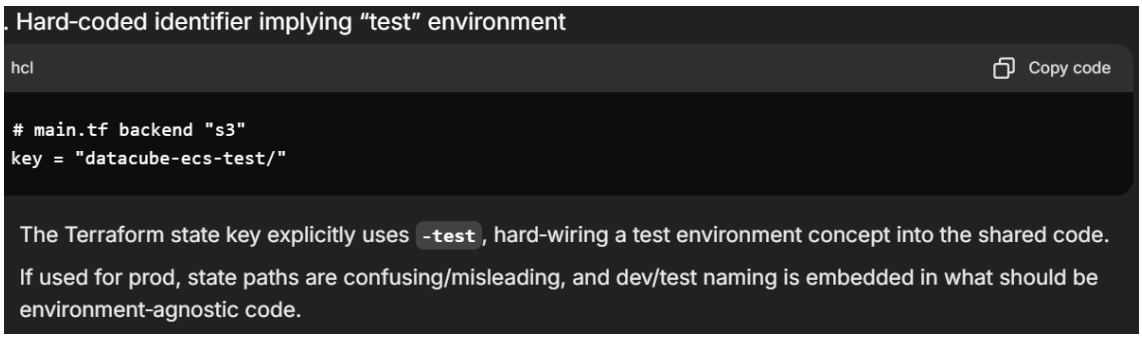

Figure 5.5: Hard-coded identifier implying “test” environment (Environment Leakage)

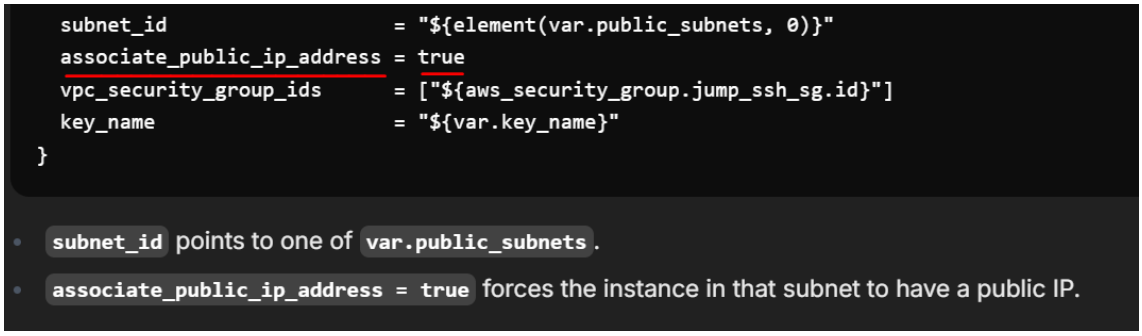

Figure 5.6: Public IP Associated with Subnet (Environment Leakage)

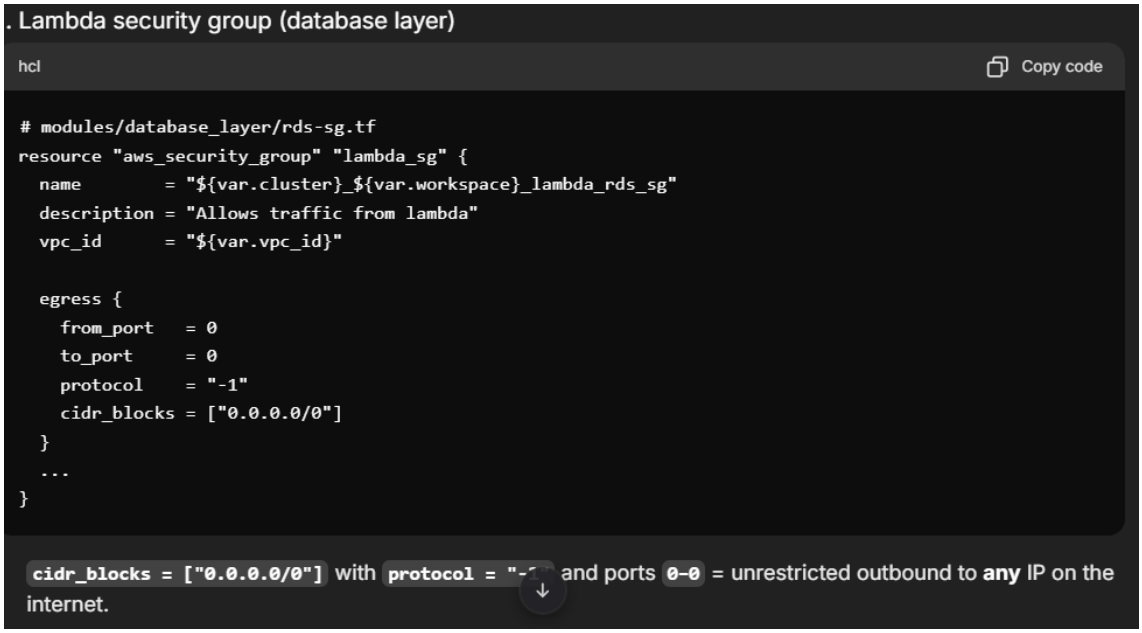

Figure 5.7: Security group rule allows egress to multiple public internet addresses (Overly Permissive Egress Rules)

6 Discussion

This chapter reflects on the main findings of this thesis and their interpretation. In addition, the discussion extends to crafting a meaningful outline for real-world deployments. An LLM-based security detection integrated into a DevOps pipeline is also proposed.

6.1 Discussion

At the core of this thesis, the experiment results show the positive potential and capabilities of LLMs as a viable replacement for rule-based tools for detecting security misconfigurations. Comparing the results from the selected LLMs, the trend observed indicates that Sonnet 4.6 gave the best outcome for both Zero-shot (0.9059) and Few-shot (0.9294) prompting. However, in terms of the usage of RAG, the results indicate minimal impact of RAG on the outcome of Few-shot prompting. The biggest beneficiary of RAG usage in terms of detection performance, was seen for GPT 5.1 and Devstral 2. The detection of misconfigurations, i.e. recall, improved from 0.5057 to 0.8605 for GPT 5.1. However, it did show over-cautiousness, i.e. the rate for false-positives shot up from Zero-shot to Few-shot prompting for GPT 5.1. This demonstrates the effectiveness of LLMs in achieving a better analytical output by extracting similar examples from a vector database.

Looking more closely, comparing the best performing rule-based tool (Tfsec) and LLM (Sonnet 4.6), the true positives for Sonnet 4.6 (82 detections out of 86) was

better than Tfsec (79 detections out of 86). This result indicates that, Sonnet 4.6 can be a good option for considerations on the basis of correct detection of misconfigurations. It should also be mentioned that, the LLMs were not provided with more customized context for each repository, like company policies or customized rules that define secure or insecure configuration, specific to a repository. This shows the robustness and the analytical capability of LLMs to infer secure/insecure configurations.

The result improvement from Zero-shot to Few-shot prompting was noticed in LLMs like Devstral 2 and GPT 5.1, which were comparatively smaller models than Sonnet 4.6. As Sonnet 4.6 is trained on larger sets of data compared to Devstral or GPT 5.1, the impact of RAG examples for better analysis had little impact. Furthermore, it should be noted that the labeled dataset that was used to populate the vector database was not controlled, meaning that the insecure misconfiguration categories were not equally distributed across the dataset. This could potentially mean that some categories may have had more examples compared to others, resulting in better test performance for categories with more similar training examples. It can also be safely said that LLMs can potentially achieve higher accuracy and F1-scores if the vector database is populated in a more context-dependent manner with respect to the repository in question.

Additionally, one major benefit of LLMs is that they provide proper explanations and suggestions for flagged insecure configurations, which is valuable assistance for administrators to make informed decisions. Furthermore, as the framework is built in a modular form, it can be integrated into a DevOps pipeline.

Figure 6.1 shows a DevOps pipeline with LLM-based security analysis integration. It visually illustrates how IaC moves through different stages, starting from code commit to deployment. In the security analysis stage, a trigger invokes the n8n workflow, where the IaC code goes through a pre-processing stage and extraction of

similar labeled examples from a pre-populated vector database. The result is funneled into an AI-agent for analysis. The outputs from these analyses are aggregated to generate a security report, which is then used in a decision gate to either allow deployment or require fixes. This figure is included to conceptually demonstrate how the proposed approach can be integrated into real-world DevOps workflows, rather than representing an actual production deployment.

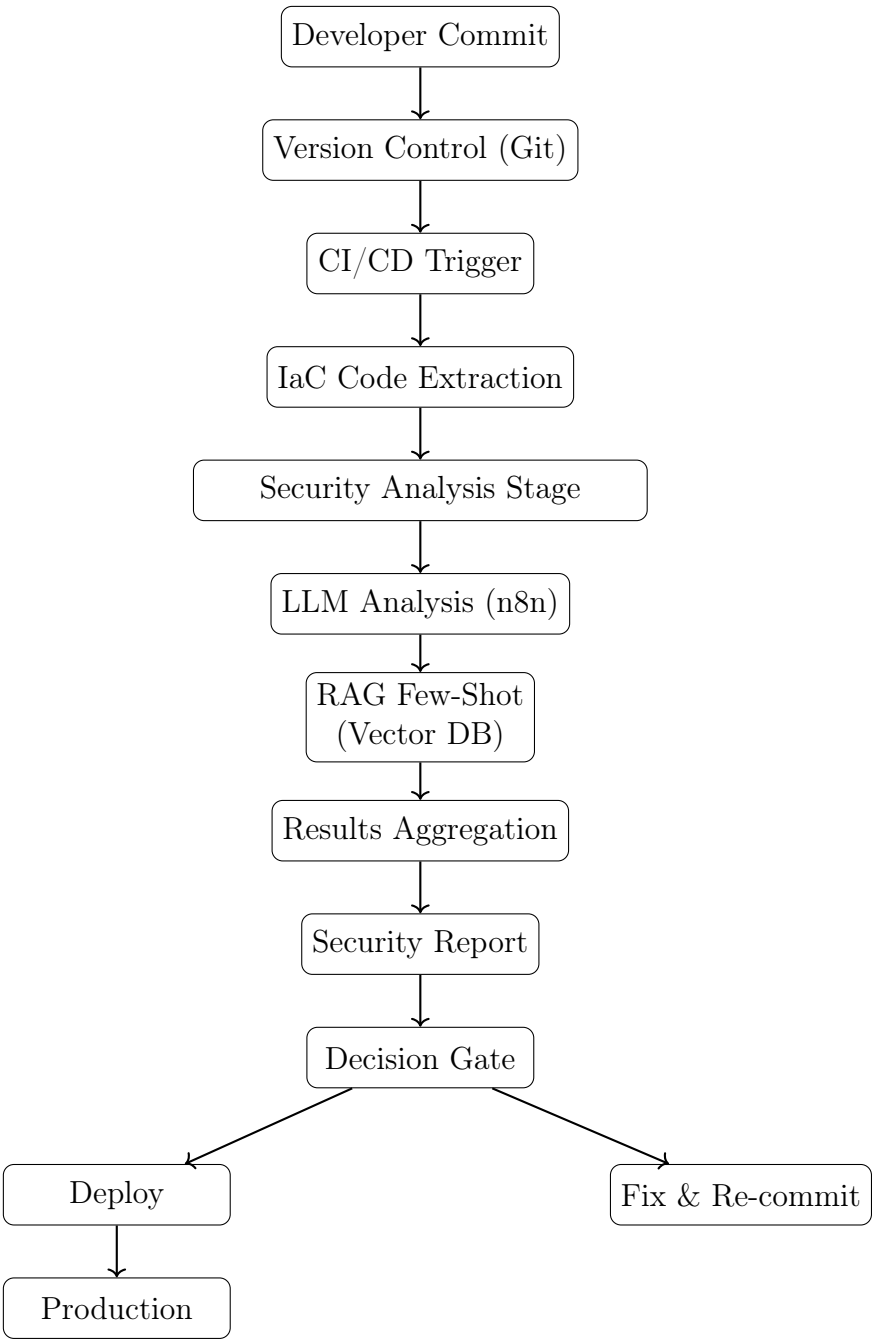

Figure 6.1: LLM-based IaC security analysis Integration in DevOps Pipeline.

7 Conclusion and Future Work

This thesis delved into an elaborate and comprehensive comparison between traditional rule-based tools and LLMs for Terraform IaC configurations of the AWS environment. The idea was to see how different LLMs performed in detecting IaC security misconfigurations, whether they can reduce false-positives and detect semantically complex misconfigurations, compared to rule-based tools.

7.1 Result of RQ1

RQ1: How do LLM-based approaches compare to rule-based static analysis tools in detecting security misconfigurations in Terraform configurations?

The experiments have shown that the LLM, Sonnet 4.6, was able to outperform established tools like Tfsec in detection accuracy. It achieved a higher true-positive count of 82 compared to 79. The experiments with RAG revealed that models like GPT 5.1 and Devstral 2 had significant gains in recall scores when provided with context-aware examples from a vector database. Hence, LLMs have shown the ability to perform on par and in some cases better than static rule-based tools. This answers the question of comparative effectiveness.

7.2 Result of RQ2

RQ2: To what extent can LLM-based security testing reduce false positives compared to rule-based IaC security scanners?

The analysis of the false positives was not promising. Although improvement has been seen for misconfiguration detection results, false-positives were either similar or higher for LLMs compared to rule-based tools. Few-shot experiments for most of the LLMs have occasionally introduced "over-cautiousness", meaning increased false-positive rates compared to Zero-shot prompting.

7.3 Result of RQ3

RQ3: How effectively do LLM-based models identify context-dependent and semantically complex security misconfigurations that are missed by rule-based tools?

The analysis of the experiment results has shown that LLMs are better at understanding semantics and context of the IaC configuration, file structures and their environments compared to rule-based tools. LLMs have exhibited better analytical capability to differentiate between secure and insecure security intent. The overall results indicate that the ability of LLMs to provide actionable explanations and remediation suggestions offers a level of developer assistance that static scanners cannot match.

7.4 Limitations

Despite the promising results obtained in this study, several limitations must be acknowledged. Firstly, the vector database used in the RAG approach was constructed from a relatively small dataset consisting of 50 secure and 50 insecure

labeled repositories, comprising a total of 1328 scripts. This may have been sufficient for initial experimentation, but the limited training set size may restrict the diversity of patterns captured by the system. In addition to this, the dataset does not contain an equal distribution of misconfiguration categories, which may lead to reduced detection performance for underrepresented categories.

Another limitation relates to the scope of the study. The experiments were conducted exclusively on Terraform-based IaC configurations within an AWS cloud environment. As a result, the findings may not generalize to other IaC tools (e.g. CloudFormation, Ansible) or cloud platforms like Azure or GCP. Furthermore, the dataset labeling process involved initial analysis using the rule-based tool tfsec, followed by human verification. This approach may introduce bias, potentially favoring tfsec and resulting in comparatively higher scoring evaluation metrics for the rule-based baseline.

From a practical system standpoint, integrating additional workflow automation tools, such as n8n along with LLM APIs, adds to the computational overhead and operational costs. This can increase the *"work in progress"* time for DevOps pipeline and consequently, questions the viability of deploying the solution in resource-constrained environments. Moreover, the quality of outputs from the LLMs depends on the quality of the vector database in the RAG-based approach. For example, if the stored examples in the database are inaccurate or poorly labeled, the LLM could provide misleading results.

Finally, security considerations also present challenges. Additional measures, such as data protection mechanisms including encryption and strict access control, should also be taken into account as IaC configurations may contain sensitive information. Failure to impose proper guardrails risks exposing confidential infrastructure information during processing. These limitations point out the potential areas of improvement and should be considered when interpreting the results of this

study.

7.5 Future Work

The work of this thesis can be further extended for future improvements. One direction could be to expand the size and diversity of the labeled dataset used to populate the vector database. Increasing the number of secure and insecure samples with balanced representation of different misconfiguration categories would likely improve the robustness of the RAG approach.

The second direction could be the broadening of scope that spans beyond Terraform and AWS environments and applying the same methodology to other cloud providers (like Azure and GCP) and Infrastructure-as-Code tools such as Ansible, Chef and Puppet. This would help assess the generalizability of the approach and determine its applicability across diverse real-world scenarios.

Another area of improvement could be data preprocessing. Delving into more advanced preprocessing techniques, such as removing irrelevant code blocks, normalizing configurations or filtering noisy data, could enhance the quality of the vector database. This, in turn, may reduce false positives and improve overall detection accuracy.

Model selection also presents an opportunity for optimization. Future work could investigate a wider range of LLMs to identify an optimal balance between performance and cost. The goal would be to select a model that is sufficiently capable of accurate analysis without incurring excessive computational or financial overhead.

Lastly, prompt engineering can be further refined to utilize the full analytical capability of LLMs. Designing more structured and context-aware prompts may improve the detection of complex and subtle misconfigurations. Applying the work conducted in this thesis along with the future work recommendations in live CI/CD

environments would assist researchers to evaluate cost and sustainability at scale. This would justify this work not only as precise but also reliable in terms of resource consumption.

To conclude, LLM-integrated DevOps pipelines represent a significant advancement in proactive infrastructure security. By bridging the gap between rigid rule sets and complex, real-world configurations, LLM-based approaches provide a more nuanced, scalable and intelligent framework for mitigating security misconfigurations. Future advancements in prompt engineering and dataset balancing will likely further solidify LLMs' case as a viable option for modern and secure DevOps practices.

References

[1] K. Morris, *Infrastructure as Code: Dynamic Systems for the Cloud Age*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2020, ISBN: 978-1-098-11467-1.

[2] M. Artac, T. Borovšak, E. Di Nitto, M. Guerriero, and D. A. Tamburri, "DevOps: Introducing infrastructure-as-code", in *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, Buenos Aires, Argentina: IEEE, 2017, pp. 497–498. DOI: 10.1109/ICSE-C.2017.162.

[3] Aqua Security. "Tfsec: Security scanner for your terraform code", Accessed: Jan. 21, 2026. [Online]. Available: https://github.com/aquasecurity/tfsec.

[4] Tenable. "Terrascan: Static code analyzer for infrastructure as code", Accessed: Jan. 21, 2026. [Online]. Available: https://github.com/tenable/terrascan.

[5] M. B. R. Amin and D. White, "Securing IaC: Comparing Checkov, Terrascan, and tfsec on AWS and Azure", in *Software Solutions Architecture and DevOps Poster Symposium*, Dublin, Ireland, 2025.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need", in *Advances in Neural Information Processing Systems*, vol. 30, Long Beach, CA, USA: Curran Associates, Inc., 2017, pp. 6000–6010.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding", in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota: Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[8] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training code representations with data flow", in *9th International Conference on Learning Representations (ICLR 2021)*, Virtual Event, Austria: OpenReview.net, 2021.

[9] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks", in *Advances in Neural Information Processing Systems*, vol. 32, Vancouver, Canada: Curran Associates, Inc., 2019, pp. 10 197–10 207.

[10] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection", in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS 2018)*, San Diego, CA, USA: The Internet Society, 2018. DOI: 10.14722/ndss.2018.23158.

[11] L. Lopez, A. Baale, and K. Prakash. "Security-aware prompt engineering for infrastructure as code automation", Accessed: Jan. 21, 2026. [Online]. Available: https://www.researchgate.net/publication/399616200_Security-Aware_Prompt_Engineering_for_Infrastructure_as_Code_Automation.

[12] A. Malhotra and M. Massimi. "Cloud security evolution: Years of progress and challenges", IBM, Accessed: Jan. 21, 2026. [Online]. Available: https://www

.ibm.com/think/insights/cloud-security-evolution-progress-and-challenges.

[13] Department of Defense, "Cloud security playbook: Volume 1", U.S. Department of Defense, Tech. Rep. Version 1.0, Feb. 2025. Accessed: Jan. 21, 2026. [Online]. Available: https://dodcio.defense.gov/Portals/0/Documents/Library/CloudSecurityPlaybookVol1.pdf.

[14] R. Croft, D. Newlands, Z. Chen, and M. A. Babar, "An empirical study of rule-based and learning-based approaches for static application security testing", in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Bari, Italy, 2021, pp. 1–12. DOI: 10.1145/3475716.3475781.

[15] G. Kim, J. Humble, P. Debois, J. Willis, and N. Forsgren, *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*, 2nd ed. Portland, OR, USA: IT Revolution Press, 2021, ISBN: 978-1-950508-40-2.

[16] K. Morris, *Infrastructure as code – NGINX preview edition*, Preview Edition, 60 pp., 2015.

[17] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective* (SEI Series in Software Engineering). Upper Saddle River, NJ, USA: Addison-Wesley Professional, 2015, ISBN: 978-0-13-404984-7.

[18] 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. 106 593, 2021. DOI: 10.1016/j.infsof.2021.106593.

[19] S. Achar, "Enterprise SaaS workloads on new-generation infrastructure-as-code (IaC) on multi-cloud platforms", *Global Disclosure of Economics and Business*, vol. 10, no. 2, pp. 55–74, 2021. DOI: 10.18034/gdeb.v10i2.652.

[20] The MITRE Corporation. "CWE top 25 most dangerous software weaknesses 2025", Common Weakness Enumeration (CWE), Accessed: Feb. 12, 2026. [Online]. Available: https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html.

[21] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts", in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Montreal, Canada: IEEE, 2019, pp. 164–175. DOI: 10.1109/ICSE.2019.00033.

[22] Joint Task Force Transformation Initiative, "Security and privacy controls for federal information systems and organizations", National Institute of Standards and Technology (NIST), Tech. Rep. NIST SP 800-53 Rev. 4, Apr. 2013, Updated January 22, 2015. Withdrawn September 23, 2021, superseded by Rev. 5. DOI: 10.6028/NIST.SP.800-53r4. Accessed: Feb. 18, 2026. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-53r4.

[23] T. Ylonen and C. Lonvick, *The secure shell (SSH) protocol architecture*, RFC 4251, IETF, Jan. 2006. DOI: 10.17487/RFC4251. Accessed: Feb. 21, 2026. [Online]. Available: https://www.rfc-editor.org/info/rfc4251.

[24] P. Mutaf, "Defending against a denial-of-service attack on TCP", in *Proceedings of the 2nd International Workshop on Recent Advances in Intrusion Detection (RAID 1999)*, ser. Lecture Notes in Computer Science, vol. 1907, West Lafayette, IN, USA: Springer, 1999.

[25] S. Kandragula, "Pegasus aircraft information breach of 2022", *Journal of Artificial Intelligence, Machine Learning & Data Science*, vol. 1, no. 4, pp. 1399–1402, 2023. DOI: 10.51219/JAIMLD/srikanth-kandragula/316.

[26] Office of the Comptroller of the Currency, "In the matter of capital one, national association: Formal agreement", U.S. Office of the Comptroller of the Currency, Tech. Rep., Aug. 2020. Accessed: Apr. 20, 2026. [Online]. Available: https://www.occ.gov/static/enforcement-actions/ea2020-056.pdf.

[27] H. Myrbakken and R. Colomo-Palacios, "Devsecops: A multivocal literature review", in *Software Process Improvement and Capability Determination*, Cham: Springer, 2017, pp. 17–29. DOI: 10.1007/978-3-319-67383-7_2.

[28] D. Shackleford, "A DevSecOps playbook", SANS Institute, Tech. Rep., Feb. 2017. Accessed: Feb. 23, 2026. [Online]. Available: https://www.sans.org/white-papers/36607/.

[29] A. Habib and M. Pradel, "How many of all bugs do we find? A study of static bug detectors", in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France: ACM, 2018, pp. 317–328. DOI: 10.1145/3238147.3238213.

[30] C. Cadar and A. F. Donaldson, "Analysing the program analyser", in *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*, Austin, TX, USA: ACM, 2016, pp. 765–768. DOI: 10.1145/2889160.2889206.

[31] A. War, A. Diallo, A. Habib, J. Klein, and T. F. Bissyandé, "Vulnerabilities in infrastructure as code: What, how many, and who?", *Empirical Software Engineering*, vol. 30, no. 5, p. 120, 2025. DOI: 10.1007/s10664-025-10672-8.

[32] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering", *ACM Computing Surveys*, vol. 54, no. 10s, p. 206, 2022. DOI: 10.1145/3505243.

[33] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design", *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994. DOI: 10.1109/32.295895.

[34] R. L. Russell, L. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning", in *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Orlando, FL, USA: IEEE, 2018, pp. 757–762. DOI: 10.1109/ICMLA.2018.00120.

[35] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs", in *2014 IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, USA: IEEE, 2014, pp. 590–604. DOI: 10.1109/SP.2014.44.

[36] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness", *ACM Computing Surveys*, vol. 51, no. 4, p. 81, 2018. DOI: 10.1145/3212695.

[37] N. Borovits, I. Kumara, P. Krishnan, S. Dalla 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)*, Virtual, USA: ACM, 2020, pp. 7–12. DOI: 10.1145/3416505.3423564.

[38] A. War, A. A. Rawass, A. K. Kaboré, J. Samhi, J. Klein, and T. F. Bissyandé, *Detection of security smells in IaC scripts through semantics-aware code and language processing*, 2025. arXiv: 2509.18790 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2509.18790.

[39] M. K. Goyal and R. Chaturvedi, "Detecting cloud misconfigurations with RAG and intelligent agents: A natural language understanding approach", *Journal of Electrical Systems*, vol. 20, no. 11s, pp. 2558–2570, 2024. DOI: 10.52783/jes.7882. Accessed: Mar. 7, 2026. [Online]. Available: https://journal.esrgroups.org/jes/article/view/7882.

[40] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software", in *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland: IEEE, 2012, pp. 837–847. DOI: 10.1109/ICSE.2012.6227135.

[41] J. Wen, Z. Chen, Z. Zhu, F. Sarro, Y. Liu, H. Ping, and S. Wang, "LLM-based misconfiguration detection for AWS serverless computing", *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 4, 110:1–110:28, 2026. DOI: 10.1145/3745766.

[42] E. Malul, Y. Meidan, D. Mimran, Y. Elovici, and A. Shabtai, *GenKubeSec: LLM-based Kubernetes misconfiguration detection, localization, reasoning, and remediation*, 2024. arXiv: 2405.19954 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2405.19954.

[43] F. Minna, F. Massacci, and K. Tuma, "Analyzing and mitigating (with LLMs) the security misconfigurations of Helm charts from Artifact Hub", *Empirical Software Engineering*, vol. 30, no. 5, p. 132, 2025. DOI: 10.1007/s10664-025-10688-0.

[44] M. M. Hassan, J. Salvador, S. K. Karmaker Santu, and A. Rahman, "State reconciliation defects in infrastructure as code", *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1865–1888, Jul. 2024. DOI: 10.1145/3660790.

[45] Y. Li, M. Grella, D. Nahmias, G. Engelberg, D. Klein, G. Guizzardi, T. van Ede, and A. Continella. "Gensiac: Toward security-aware infrastructure-as-code generation with large language models". CoRR, vol. abs/2511.12385, Preprint. eprint: arXiv:2511.12385. [Online]. Available: https://arxiv.org/abs/2511.12385.

[46] G. C. Kakaraparthi, "A comparative study of LLMs for infrastructure-as-code generation and optimization", *Computer Fraud and Security*, 2022. DOI: 10.52710/cfs.730. [Online]. Available: https://computerfraudsecurity.com/index.php/journal/article/view/730.

[47] C. Gunawat and A. Khanna, "AI-enhanced infrastructure as code (IaC) for smart configuration management", *Electro Sphere Electronic Engineering Bulletin*, vol. 1, no. 2, pp. 1–7, 2025. DOI: 10.2139/ssrn.5225391. [Online]. Available: https://ssrn.com/abstract=5225391.

[48] G. De Vito, F. Palomba, and F. Ferrucci, "SecLLM: Enhancing security smell detection in IaC with large language models", *IEEE Access*, vol. 13, pp. 204 480–204 498, 2025. DOI: 10.1109/ACCESS.2025.11269824.

[49] S. Reis, R. Abreu, M. d'Amorim, and D. Fortunato, "Leveraging practitioners' feedback to improve a security linter", in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*, Rochester, MI, USA: ACM, Oct. 2022, 66:1–66:12. DOI: 10.1145/3551349.3560419.

[50] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in Ansible and Chef scripts: A replication study", *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 1, pp. 1–31, Jan. 2021. DOI: 10.1145/3408897.

[51] N. Saavedra and J. F. Ferreira, "GLITCH: Automated polyglot security smell detection in infrastructure as code", in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*, Rochester, MI, USA: Association for Computing Machinery, 2022, pp. 1–12. DOI: 10.1145/3551349.3556945.

[52] R. Opdebeeck, A. Zerouali, and C. De Roover, "Control and data flow in security smell detection for infrastructure as code: Is it worth the effort?", in *Proceedings of the IEEE/ACM 20th International Conference on Mining Software Repositories (MSR 2023)*, Melbourne, Victoria, Australia: IEEE/ACM, May 2023, pp. 534–545. DOI: 10.1109/MSR59073.2023.00079.

[53] R. Y. Reyes, B. M. Ampel, and H. Chen, "Large language models for infrastructure as code vulnerability remediation", in *Pre-ICIS Workshop on Information Security and Privacy (WISP 2025)*, Nashville, TN, USA, Dec. 2025.

[54] galcan, *Terraform_sec: Terraform security dataset*, https://huggingface.co/datasets/galcan/terraform_sec, Accessed: 2026-02-10. Hugging Face dataset. License: Apache 2.0, 2024.

[55] M. Zheng, J. Pei, L. Logeswaran, M. Lee, and D. Jurgens, "When "A helpful assistant" is not really helpful: Personas in system prompts do not improve performances of large language models", in *Findings of the Association for Computational Linguistics: EMNLP 2024*, Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024. [Online]. Available: https://aclanthology.org/2024.findings-emnlp.888.

[56] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm", in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems (CHI EA '21)*, Yokohama, Japan: ACM, May 2021, 314:1–314:7. DOI: 10.1145/3411763.3451760.

[57] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, *Retrieval-augmented generation for large language models: A survey*, 2023. arXiv: 2312.10997 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2312.10997.

[58] V. Gummadi, P. Udayaraju, V. R. Sarabu, C. Ravulu, D. R. Seelam, and S. Venkataramana, "Enhancing communication and data transmission security in RAG using large language models", in *Proceedings of the 4th International Conference on Sustainable Expert Systems (ICSES 2024)*, Kaski, Nepal, 2024, pp. 612–617. DOI: 10.1109/ICSES63445.2024.10763024.

Appendix A Mapping Misconfiguration Category to Rule-ID

A.1 Publicly Exposed Storage (S3 / Blob)

Rule-ID	Description
AVD-AWS-0053	Load balancer publicly exposed or not restricted as required.
AVD-AWS-0086	Network exposure: security groups/endpoints allow 0.0.0.0/0.
AVD-AWS-0087	RDS instance publicly accessible or weakly restricted.
AVD-AWS-0090	RDS enhanced/required monitoring not enabled.
AVD-AWS-0091	S3 account-level “IgnorePublicAcls” not enabled.
AVD-AWS-0093	S3 bucket “BlockPublicPolicy” not enabled.
AVD-AWS-0094	S3 bucket “BlockPublicAcls” not enabled.
AVD-AWS-0164	Lambda environment contains plaintext secrets / not using KMS.

Table A.1: Violation class description 1

A.2 Overly Permissive IAM Policies

Rule-ID	Description
AVD-AWS-0057	IAM policy document uses sensitive action 'logs:CreateLogGroup' on wildcarded resource '*'
AVD-AWS-0062	IAM Password policy should have expiry less than or equal to 90 days.
AVD-AWS-0104	Security group rule allows egress to multiple public internet addresses.
AVD-AWS-0123	Multi-Factor authentication is not enforced for group.
AVD-AWS-0143	One or more policies are attached directly to a user

Table A.2: Violation class description 2

A.3 Open Security Group Ingress (0.0.0.0/0)

Rule-ID	Description
AVD-AWS-0107	Security group rule allows ingress from public internet.

Table A.3: Violation class description 3

A.4 Missing Encryption at Rest

Rule-ID	Description
AVD-AWS-0008	Root block device is not encrypted.
AVD-AWS-0023	Table encryption is not enabled.
AVD-AWS-0025	Table encryption does not use a customer-managed KMS key.
AVD-AWS-0026	EBS volume is not encrypted.
AVD-AWS-0027	EBS volume does not use a customer-managed KMS key.
AVD-AWS-0064	Stream does not use KMS encryption.
AVD-AWS-0088	Bucket does not have encryption enabled.
AVD-AWS-0131	Root block device is not encrypted.
AVD-AWS-0132	Bucket does not encrypt data with a customer managed key.

Table A.4: Violation class description 4

A.5 Secrets Hardcoded in Code

Rule-ID	Description
AVD-AWS-0101	Hardcoded credentials in config.

Table A.5: Violation class description 5

A.6 Logging / Monitoring Disabled

Rule-ID	Description
AVD-AWS-0010	Distribution does not have logging enabled.
AVD-AWS-0024	Point-in-time recovery is not enabled.
AVD-AWS-0034	Cluster does not have container insights enabled.
AVD-AWS-0077	Instance has very low backup retention period.
AVD-AWS-0089	Bucket does not have logging enabled.
AVD-AWS-0090	Bucket does not have versioning enabled.
AVD-AWS-0177	Instance does not have Deletion Protection enabled.
AVD-AWS-0178	VPC Flow Logs is not enabled for VPC.

Table A.6: Violation class description 6

A.7 Overly Permissive Egress Rules

Rule-ID	Description
AVD-AWS-0104	Security group rule allows egress to multiple public internet.

Table A.7: Violation class description 7

A.8 Environment Leakage (Dev Config in Prod)

Rule-ID	Description
AVD-AWS-0011	Distribution does not utilise a WAF.
AVD-AWS-0012	Distribution allows unencrypted communications.
AVD-AWS-0013	Distribution allows unencrypted communications.
AVD-AWS-0028	Instance does not require IMDS access to require a token.
AVD-AWS-0164	Subnet associates public IP address.

Table A.8: Violation class description 8

Appendix B Pre-processing and Workflow Scripts

B.1 Extraction of scripts from JASONL formatted dataset

Listing B.1: Extraction of scripts from JASONL formatted dataset.

```
import json
import re
import os

def extract_first_100(input_file, output_file, label):
    with open(input_file, "r", encoding="utf-8") as infile, \
         open(output_file, "w", encoding="utf-8") as outfile:

        for i, line in enumerate(infile):
            if i >= 100:
                break

            project = json.loads(line)

            project_id = project.get("project_id", f"project_{i}")
            terraform_input = project.get("input", "")

            outfile.write("="*80 + "\n")
            outfile.write(f"PROJECT_INDEX: {i}\n")
            outfile.write(f"PROJECT_ID: {project_id}\n")
            outfile.write(f"LABEL: {label}\n")
            outfile.write("="*80 + "\n\n")

            # Extract file sections
            # Pattern: ### folder\filename.tf
            file_sections = re.split(r"### ", terraform_input)

            for section in file_sections:
                if ".tf" in section:
                    lines = section.split("\n", 1)
```

```
                    filename = lines[0].strip()
                    content_block = lines[1] if len(lines) > 1 else ""

                    # Extract code inside ```terraform ```
                    code_match = re.search(r"```terraform\n(.*?)```",
    content_block, re.DOTALL)

                    if code_match:
                        code = code_match.group(1)

                        outfile.write(f"--- FILE: {filename} ---\n")
                        outfile.write(code.strip() + "\n\n")

            outfile.write("\n\n")

    print(f"Finished extracting first 100 entries from {input_file}")


extract_first_100(
    input_file="insecure_projects.jsonl",
    output_file="insecure_first_100.txt",
    label="INSECURE"
)

extract_first_100(
    input_file="secure_projects.jsonl",
    output_file="secure_first_100.txt",
    label="SECURE"
)
```

B.2 Code for separating each test repository

Listing B.2: Code for separating each test repository.

```
import json
import re
import os

BASE_OUTPUT_DIR = "terraform_dataset"


def reconstruct_projects(input_file, label, limit=100):
    label_dir = os.path.join(BASE_OUTPUT_DIR, label.lower())
    os.makedirs(label_dir, exist_ok=True)

    with open(input_file, "r", encoding="utf-8") as infile:
        for i, line in enumerate(infile):
            if i >= limit:
                break

            project = json.loads(line)

            project_id = project.get("project_id", f"project_{i}")
            terraform_input = project.get("input", "")
```

```
            project_dir = os.path.join(label_dir, project_id)
            os.makedirs(project_dir, exist_ok=True)

            # Split sections by file marker
            file_sections = re.split(r"### ", terraform_input)

            for section in file_sections:
                if ".tf" in section:
                    lines = section.split("\n", 1)
                    filename = lines[0].strip()
                    content_block = lines[1] if len(lines) > 1 else ""

                    # Extract terraform code block
                    code_match = re.search(r"```terraform\n(.*?)```",
    content_block, re.DOTALL)

                    if code_match:
                        code = code_match.group(1).strip()

                        # Normalize Windows-style paths
                        filepath = filename.replace("\\", os.sep)

                        full_path = os.path.join(project_dir, filepath)

                        # Create subdirectories if needed
                        os.makedirs(os.path.dirname(full_path), exist_ok=
    True)

                        # Write .tf file
                        with open(full_path, "w", encoding="utf-8") as
    tf_file:
                            tf_file.write(code)

    print(f"Finished reconstructing first {limit} projects from {input_file
    }")


# Run reconstruction
reconstruct_projects("secure_projects.jsonl", label="SECURE", limit=50)
reconstruct_projects("insecure_projects.jsonl", label="INSECURE", limit=50)
```

B.3 Splitting of test repositories in text files

Listing B.3: Splitting of test repositories in text files.

```
import os
import re

# INPUT FILES
input_files = [
    "secure_first_100.txt",
    "insecure_first_100.txt"
]

# OUTPUT DIRECTORY
```

```
output_dir = "terraform_projects_split"
os.makedirs(output_dir, exist_ok=True)

project_counter = 1

for file_path in input_files:
    with open(file_path, "r", encoding="utf-8") as f:
        content = f.read()

    # Split projects using the separator
    projects = re.split(r"=+\s*\nPROJECT_INDEX:", content)

    # The first split may contain empty text
    for project in projects:
        project = project.strip()
        if not project:
            continue

        # Add back the PROJECT_INDEX label if removed
        project_text = "PROJECT_INDEX:" + project

        # Extract project id if possible
        match = re.search(r"PROJECT_ID:\s*(.+)", project_text)
        if match:
            project_id = match.group(1).strip()
        else:
            project_id = f"project_{project_counter}"

        # Clean filename
        project_id = project_id.replace("/", "_").replace("\\", "_")

        output_file = os.path.join(output_dir, f"{project_id}.txt")

        with open(output_file, "w", encoding="utf-8") as out:
            out.write(project_text)

        project_counter += 1

print(f"Finished. {project_counter-1} project files created in '{output_dir
    }'")
```

B.4 Code for Binary to Pre-processed JSON

Listing B.4: Code for Binary to Pre-processed JSON.

```
// Iterate through items from Read Files node
return $input.all().map(item => {
  const content = item.json.data;

  // Extract label
  const matchLabel = content.match(/LABEL:\s*(SECURE|INSECURE)/i);
  const label = matchLabel ? matchLabel[1].toUpperCase() : "UNKNOWN";

  // Extract project ID
  const matchProjectID = content.match(/PROJECT_ID:\s*([^\s]+)/i);
```

```
  const projectId = matchProjectID ? matchProjectID[1] : "unknown_project";

  // Extract file name
  const matchFile = content.match(/--- FILE:\s*([^\n]+)\s*---/i);
  const fileName = matchFile ? matchFile[1].trim() : "unknown_file";

  // Remove dataset header
  const cleanedCode = content.replace(/PROJECT_INDEX[\s\S]*?={10,}/, '').
    trim();

  // Combine code with metadata inside `data`
  const combinedData = {
    code: cleanedCode,
    label: label,
    projectId: projectId,
    fileName: fileName
  };

  return {
    json: {
      data: combinedData
    }
  };
});
```

B.5 Code for Creating Chunks for Vector Database

Listing B.5: Code for Creating Chunks.

```
return $input.all().flatMap(item => {

  const text = item.json.data.code; // Terraform code
  const label = item.json.data.label;
  const projectId = item.json.data.projectId;
  const fileName = item.json.data.fileName;

  const MAX_CHUNK_SIZE = 200; // max characters per chunk

  // Match Terraform resource blocks
  const resourceRegex = /resource\s+"([^"]+)"\s+"([^"]+)"\s*{[\s\S]*?}/g;
  const matches = [...text.matchAll(resourceRegex)];

  const allChunks = [];

  matches.forEach((match, resourceIndex) => {
    const resourceType = match[1];
    const resourceName = match[2];
    const block = match[0];

    // Split block into sub-chunks if too long
    for (let i = 0; i < block.length; i += MAX_CHUNK_SIZE) {
      const subBlock = block.slice(i, i + MAX_CHUNK_SIZE);

      const documentCode = [
        `LABEL: ${label}`,
```

```
        `PROJECT_ID: ${projectId}`,
        `FILE_NAME: ${fileName}`,
        `RESOURCE_TYPE: ${resourceType}`,
        `RESOURCE_NAME: ${resourceName}`,
        `CHUNK_INDEX: ${resourceIndex}.${Math.floor(i / MAX_CHUNK_SIZE)}`,
        '',
        subBlock
      ].join('\n');

      allChunks.push({
        json: {
          document: {
            code: documentCode
          }
        }
      });
    }
  });

  return allChunks;
});
```

B.6 Code for Chunking During Security Testing

Listing B.6: Code for Chunking.

```
return $input.all().flatMap(item => {

  const text = item.json.data.code; // Terraform code

  const MAX_CHUNK_SIZE = 200; // max characters per chunk

  // Match Terraform resource blocks
  const resourceRegex = /resource\s+"([^"]+)"\s+"([^"]+)"\s*{[\s\S]*?}/g;
  const matches = [...text.matchAll(resourceRegex)];

  const allChunks = [];

  matches.forEach((match, resourceIndex) => {
    const resourceType = match[1];
    const resourceName = match[2];
    const block = match[0];

    // Split block into sub-chunks if too long
    for (let i = 0; i < block.length; i += MAX_CHUNK_SIZE) {
      const subBlock = block.slice(i, i + MAX_CHUNK_SIZE);

      const documentCode = [
        `RESOURCE_TYPE: ${resourceType}`,
        `RESOURCE_NAME: ${resourceName}`,
        `CHUNK_INDEX: ${resourceIndex}.${Math.floor(i / MAX_CHUNK_SIZE)}`,
        '',
        subBlock
      ].join('\n');
```

```
      allChunks.push({
        json: {
          document: {
            code: documentCode
          }
        }
      });
    }
  });

  return allChunks;
});
```