INFORMATICS INSTITUTE OF TECHNOLOGY

INFORMATICS INSTITUTE OF TECHNOLOGY

In Collaboration with

UNIVERSITY OF WESTMINSTER

Cryptographic Signing and Verification of Terraform Plans for Supply Chain Integrity

A Project Proposal by

J.R.M.S.B.Karunarathne

Supervised by

Mr. Danishka Navin

Abstract

Problem: There is infrastructure as Code (IaC) tools that can create rapid, automatic deployment into clouds like Terraform that provide new security threats. Hackers can interfere with Terraform modules or configuration files to introduce malicious code and gain access to full environments (Schneeweisz, 2022; Sigstore, 2023). Current defences are based on intense use of manual reviews and statical analysis, which would not ensure the integrity and authenticity of codes. The proposed project solves that issue by implementing a cryptographic provenance system, which guarantees the reliability of Terraform IaC artifacts with the help of digital signatures and attestable verifications.

Methodology: The solution that is proposed combines Sigstore (Cosign) and in-toto provenance solutions into Terraform patterns. To ensure that Terraform modules are signed automatically, and those signatures must be captured in transparency logs, the deployment pipeline is instrumented. An experimental tool has been created to create and check cryptographic signatures on every deployment stage. The methodology is based on the quantitative (measurement of the accuracy of attack detection and the runtime overhead) and qualitative (cases studies of real life supply chain incidents) analysis. The effectiveness of the system is compared to the existing baseline practices, which include Git hash pinning and manual checking of module verification.

Subject Descriptors (ACM CCS):
Software and its engineering → Software creation and management → Software development techniques → DevOps
Security and privacy → Software and application security → Software supply chain security

Keywords:
Infrastructure as Code, Terraform, Software Supply Chain Security, Cryptographic Provenance, Digital Signatures, DevSecOps

Table of Content

CHAPTER 01: INTRODUCTION 1

1.1. Chapter Overview 1

1.2. Problem Background 1

1.3. Problem Definition 3

1.3.1. Research Problem Statement 4

1.4. Research Motivation 4

1.5. Existing Work 5

1.6. Research Gap 7

1.7. Contribution to Knowledge 7

1.7.1. Contribution to the Problem Domain 7

1.7.2. Research Domain Contribution (Technological Contribution) 8

1.8. Research Challenges 8

1.9. Research Questions 10

1.10. Research Aim 11

1.11. Research Objectives 11

1.12. Hardware/Software Requirements 14

1.13. Chapter Summary 16

CHAPTER 02: LITERATURE REVIEW 17

2.1. Chapter Overview 17

2.2. Problem Domain 17

2.3. Existing Work 18

2.3.1 IaC Security Practices 19

2.3.2 Supply Chain Security for Terraform 20

2.3.3 Cryptographic Provenance and Attestations 21

2.3.4 Related Technologies 22
2.3.5 Critique of Existing Approaches 23
2.4. Dataset Selection 23
2.5. Benchmarking and Evaluation 24
2.6. Chapter Summary 25
CHAPTER 03: METHODOLOGY 27
3.1 Chapter Overview 27
3.2 Research Methodology 27
3.3 Development Methodology 29
3.3.1 Requirements Analysis 29
3.3.2 System Design 29
3.3.3 Implementation Steps 31
3.3.4 Testing and Evaluation 33
3.4 Project Management 34
3.4.1 Scope and Deliverables 34
3.4.2 Work Plan and Schedule 34
3.4.3 Risk Analysis 35
3.5 Chapter Summary 36
CHAPTER 04: Software Requirement Specification (SRS) 37
4.1 Chapter Overview 37
4.2 Rich Picture Diagram 38
4.3 Stakeholder Analysis 38
4.3.1 Stakeholder Onion Model 38
4.3.2 Stakeholder Viewpoints 40
4.4 Selection of Requirement Elicitation Methodologies 41
4.4.1 Literature Review 41

4.4.2 Surveys....42
4.4.3 Brainstorming and Observation. 42
4.5 Discussion of Findings.... 43
4.5.1 Literature Review Findings....43
4.5.2 Survey Findings 44
4.5.3 Response Analysis 44
4.5.4 Brainstorming/Observation....53
4.6 Summary of Findings.... 54
4.7 Context Diagram.... 55
4.8 Use Case Diagram.... 55
4.9 Use Case Descriptions 56
4.10 Requirements 58
4.10.1 Functional Requirements....58
4.10.2 Non-Functional Requirements....58
4.11 Chapter Summary 60
References……..... 62
Appendix……..... 64

List of Figures

Figure 1: Current Architecture Diagram (self-composed).... 3
Figure 2: Architecture Diagram (Self-composed) 31
Figure 3: Component Diagram (Self-composed).... 33
Figure 4: Work Plan (Self-composed) 35
Figure 5: Rich Picture Diagram (Self-composed) 38
Figure 6: Stakeholder Onion Model (Self-composed).... 39
Figure 7: Context Diagram (Self-composed).... 55
Figure 8: Use case Diagram (Self-composed) 56
Figure 9: Gantt Chart(Self-composing) 61

List of Tables

Table 1: Existing Work 6
Table 2: Research Objectives 13
Table 3: Table of Software and Hardware Requirements 16
Table 4: Existing Work with Relevance 19
Table 5: Evaluation Metrics Comparison 25
Table 6: The layers of Saunders' Research Onion 29
Table 7: Risk Analysis 36
Table 8: Stakeholder Viewpoints 41
Table 9: Literature Review Findings 44
Table 10: Response Analysis of the Survey 53
Table 11: Brainstorming Findings 54
Table 12: Summary of Findings 54
Table 13: Use Case Descriptions UC:01 57
Table 14: Functional Requirements 58
Table 15: Non-Functional Requirements 60
Table 16: Use Case Descriptions UC:02 64
Table 17: Use Case Descriptions UC:03 65
Table 18: Use Case Descriptions UC:04 66
Table 19: Use Case Descriptions UC:05 67
Table 20: Use Case Descriptions UC:06 68
Table 21: Use Case Descriptions UC:07 69

List of Abbreviations

- CI/CD: Continuous Integration/Continuous Deployment
- DevSecOps: Development, Security, and Operations
- SLSA: Supply chain Levels for Software Artifacts
- SBOM: Software Bill of Materials
- TPM: Trusted Platform Module (in cryptographic systems)

CHAPTER 01: INTRODUCTION

1.1. Chapter Overview

This chapter introduces the research topic of securing Terraform based IaC using cryptographic provenance. It provides background on why Terraform and IaC security are important, and then defines the specific security problem. This outlines the motivation for the research, review existing work, and identify the gap in current knowledge. Then state the contributions, discuss key research challenges, and present the research questions. Finally, define the aim and objectives of the project. The chapter concludes with a summary.

Recent developments in software supply chain security offer new approaches that have not yet been applied to IaC. For instance, the Sigstore project provides free tools like Fulcio and Rekor, which issue short lived certificates and maintain public transparency logs for software artifacts. The SLSA framework similarly codifies provenance levels for tamper resistant builds. Applying these advances to Terraform could close a critical gap between DevSecOps theory and IaC practice. For example, one analysis found that 64% of Terraform modules in public registries contain at least one high severity misconfiguration (Eisenkot, 2022), illustrating how widespread and dangerous such issues are. This suggests that strong, automated verification is needed to ensure IaC integrity.

1.2. Problem Background

Infrastructure as Code (IaC) is a DevOps practice that uses declarative scripts to provision and manage cloud infrastructure. Terraform is a widely used IaC tool that automates the deployment of networks, servers, and services by applying a desired state configuration (Lepiller *et al.*, 2021). Treating infrastructure definitions as version controlled code enables repeatable builds and rapid iteration. However, like any code, IaC can hide misconfigurations, embedded secrets, or even malicious payloads that may compromise the resulting infrastructure (Verdet *et al.*, 2023).

Modern IaC workflows also rely on external dependencies (Terraform modules and providers) fetched from public or private registries. These dependencies expand the attack surface: if a

Terraform module or provider is tampered with or maliciously updated, every deployment using it is at risk. Industry reports identify dependency chain abuse and pipeline poisoning as top CI/CD security risks (OWASP, 2023). In Terraform's context, such risks include module sources changing unexpectedly (through repository hijacking or tag manipulation) or CI pipelines loading unverified artifacts. Recent incidents like the SolarWinds and Codecov breaches show how compromised build systems and package repositories can propagate malware widely. Well known supply chain attacks like SolarWinds and Codecov affected thousands of organizations (Tamanna *et al.*, 2024), highlighting the urgent need for stronger safeguards in IaC pipelines.

Terraform's flexibility comes with significant trade offs. Its reliance on external modules, shared registries, and state files introduces critical supply chain vulnerabilities. In one study, a large majority of Terraform Registry modules were found to contain high or critical severity security issues (Eisenkot, 2022). These vulnerabilities could be exploited by malicious actors. Attackers have weaponized widely used open source components to create systemic weaknesses (for example, the XZ Utils backdoor) and have poisoned software dependencies at scale. Consequently, frameworks like SLSA advocate for provenance tracking and artifact verification as best practices. Implementing SLSA practices such as verifying source integrity and tracking signed build metadata is believed to reduce the risk of supply chain attacks (Tamanna *et al.*, 2024).

Despite these developments, practical use of cryptographic provenance in Terraform workflows remains rare. The OWASP Top 10 CI/CD Risks report explicitly calls for "cryptographic validation of artifacts throughout the pipeline" (OWASP, 2023). Yet no mainstream IaC toolchain currently provides end to end guarantees of code origin. Traditional controls (manual reviews, version pinning, etc.) are often insufficient against skilled adversaries. For example, researchers demonstrated that typical Terraform workflows allow undetected module tampering between the plan and apply phases (Proulx, 2023). In summary, Terraform's declarative model and ecosystem yield a powerful but vulnerable supply chain, requiring novel solutions that can verify module authenticity and integrity at every step.

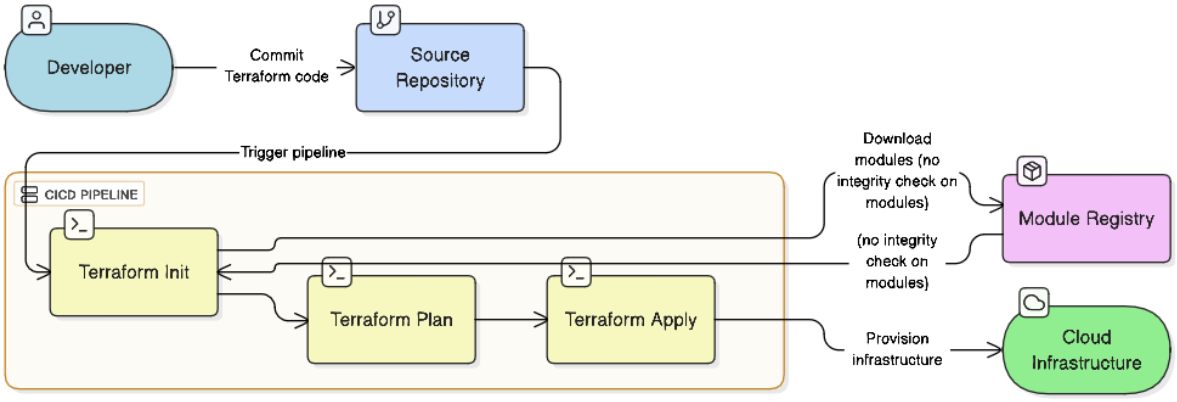


Figure 1: Current Architecture Diagram (self-composed)

1.3. Problem Definition

The core problem addressed by this research is how to ensure the integrity and authenticity of Terraform infrastructure code. Specifically, aiming to create a system where each Terraform module or provider is accompanied by verifiable metadata (digital signatures and attestations) proving its source and contents, so that any tampering becomes evident. Current practices like version pinning or manual code review are not sufficient against sophisticated supply chain attacks (Proulx, 2023). Goal is to design, implement, and evaluate a cryptographic framework for Terraform that secures the IaC supply chain end to end.

Terraform's architecture has specific points that attackers can exploit in supply chain attacks. Modules and providers are often fetched by version tag, without inherent cryptographic validation. As BoostSecurity noted, Terraform modules lack a built in signature or hash mechanism, so an attacker could alter a tagged release without detection (Proulx, 2023). Even Terraform's existing .terraform.lock.hcl lockfile only pins provider binaries (with their SHA-256 hashes) and does not lock module content. This means a user reviewing a Terraform plan cannot be sure that a subsequent terraform apply is using the same module source as the plan. A module's content could change in the interim (Proulx, 2023). This gap allows simple repo jacking or tag poisoning attacks, as described in recent threat analyses (Proulx, 2023).

In summary, Ho can introduce cryptographic verification into Terraform pipelines so that unauthorized module changes are automatically detected or prevented? This involves determining where to attach signatures or attestations in the workflow, how to integrate them with existing tools (Terraform CLI, CI/CD pipelines, etc.), and how to manage the necessary cryptographic keys and policies. Solving this problem would greatly improve the security posture of Terraform deployments and address the trust gap currently exploited by supply chain attackers.

1.3.1. Research Problem Statement

Can cryptographic provenance methods (such as digital signatures, transparency logs, and step attestations) be integrated throughout the Terraform development and deployment workflow to ensure end to end integrity and authenticity of IaC artifacts, without unduly impacting developer productivity?

1.4. Research Motivation

Securing the software supply chain has become a priority in both industry and academia. High-profile incidents have shown that trust in code delivery is fragile: attackers can exploit any weak link. The pace of modern DevOps means IaC tools like Terraform are now central to cloud infrastructure management, so any breach in their pipeline can have a large impact. However, current defense approaches are not well adapted to the nature of IaC. Traditional code signing schemes (used for compiled software) do not extend well to the versioned modules or configuration files that Terraform employs. Meanwhile, new cryptographic frameworks for supply chain security have matured (e.g. Sigstore's certificate issuance, in-toto's step attestations, and the SLSA assurance levels) and are being applied to other parts of the software stack. For example, Sigstore's Cosign can sign container images and binaries with short lived keys, and its transparency log (Rekor) makes those signatures publicly auditable. In March 2023, the Node.js npm registry added Sigstore based provenance attestations for each new package release, providing a signed record of origin (Sigstore, 2023). These advances demonstrate that strong provenance is feasible to integrate into CI/CD processes.

Applying similar methods to IaC is a natural next step. Terraform modules and providers are essentially software artifacts that could be signed and verified. Ensuring that every Terraform plan, module, and apply step carries a verifiable signature would significantly raise the bar for attackers.

For example, integrating Cosign into Terraform pipelines could enable automatic signing of module packages, and using in-toto we could capture a signed "link" file that records all module hashes used during a terraform apply. As supply chain researchers observe, implementing source integrity verification and provenance tracking can greatly reduce supply chain attack risk (Tamanna *et al.*, 2024). Notably, HashiCorp and others have begun acknowledging these issues. As of January 2024, the Terraform Registry now links published modules to immutable Git commit hashes (HashiCorp, 2024). However, this registry feature is optional and limited. It does not cover self hosted modules and does not verify the entire pipeline.

In summary, there is strong motivation to adopt cryptographic guarantees in IaC workflows. This research is driven by the opportunity to bring proven supply chain security techniques into the Terraform domain, closing known gaps and enabling secure DevSecOps practices. By achieving this, we will bridge the gap identified in HashiCorp's own advisories (HashiCorp, 2024) and address concerns highlighted in recent studies and expert blogs.

1.5. Existing Work

Citation	Summary	Limitation	Contribution
Verdet et al. (2023)	Empirical study of 812 open source Terraform projects. analyzed adherence to IaC security best practices.	Focused on configuration best practices, not supply chain provenance.	Highlighted common security gaps like weak encryption and inconsistent access control.
Schneeweisz (2022)	Demonstrated Terraform module tampering via hijacked Git tags and mutable versions.	Relied on manual mitigations like pinning commits, not automated or enforced security.	Revealed lack of cryptographic integrity in Terraform modules and proposed basic mitigations.

Sigstore (2023)	Open source framework for artifact signing and verification using Cosign and Rekor.	Not directly applied to Terraform or IaC workflows yet.	Established verifiable signing infrastructure for software artifacts.
Sirish & Kennedy (2023) / in-toto	Defined end to end cryptographic attestation for build processes.	Artifact agnostic, lacks Terraform-specific adaptation.	Offered reusable model for verifiable provenance tracking in supply chains.
Tamanna et al. (2024) / SLSA	Proposed multi-level supply chain assurance framework emphasizing provenance attestations.	Conceptual framework, not integrated with IaC tooling.	Provided structured maturity model for supply chain security.

Table 1: Existing Work

Prior research on IaC security has largely focused on detecting misconfigurations rather than ensuring supply chain provenance. For example, Verdet *et al.* (2023) conducted an empirical study of 812 open source Terraform projects. They found that some best practices (such as strict access controls) are commonly followed, while others (notably encryption at rest) are often neglected. Tools like Checkov or TFLint perform static analysis to catch common Terraform misconfigurations (e.g. hardcoded credentials, open ports), but they do not address module authenticity. OWASP's IaC security guidelines emphasize secrets management and least privilege, but they do not cover supply chain threats. Similarly, commercial guides for Terraform security (e.g. from Sysdig) advise avoiding secrets in code and using secure backends, but they stop short of recommending cryptographic verification (Douglas, 2023).

In summary, most IaC security work treats it as a configuration problem. However, recent incidents show that the supply chain of IaC code is equally critical. Even perfectly written Terraform code can be subverted by compromised dependencies. The OWASP Top 10 CI/CD Risks explicitly includes *Dependency Chain Abuse* and *Pipeline Poisoning*, warning that attackers may manipulate external dependencies to run malicious code in build environments (OWASP, 2023). In Terraform's context, every terraform apply might fetch remote modules; if an attacker can slip malicious changes into a module between validation and deployment, they can hijack the infrastructure.

1.6. Research Gap

The literature review also indicates that existing IaC security techniques are mostly cryptographic, agnostic and reactive. Static analysis or policy as code tools can detect some insecure configurations, but cannot protect against a malicious supply chain adversary who compromises dependencies. The cutting edge of supply chain security (Sigstore signing, transparency logs, in-toto attestations, SLSA provenance levels) has not been brought to integrate with Terraform workflows. In short, Terraform users still have no mechanism to associate provenance metadata with modules, plans and applies. No framework which is available has support for verifying signatures on Terraform modules when deployed, or that can add a cryptographic attestation to the pipeline of running Terraform. Thus there is a clear need chance to address this gap by building a cryptographic provenance solution for Terraform.

Which cryptographic primitives (digital signatures, transparency logs, attestation records) can be used for Terraform without causing considerable overhead? How to integrate those into your developer tools (git commit hooks, Terraform plugins?), etc. And how does such an approach stack up in terms of security guarantees and usability versus what have today (manual review, module version pinning, et al.?) By answering these questions., aim is to enable Terraform deployments that are secure by design rather than by chance.

1.7. Contribution to Knowledge

1.7.1. Contribution to the Problem Domain

This study will significantly improve the security of Infrastructure as Code (IaC) flows, including Terraform.The project has extended the signing/verification process to terraform modules/configurations by implementing prototype of extension. It shows that cryptographic provenance which previously only fits software artifacts can be effectively leveraged for IaC environments. Formal threat modeling and security evaluation will supply quantifiable evidence as to how provenance mitigates threats such as module compromise and dependency poisoning. The guidelines and best practices that emerge from this work will enable DevOps teams to integrate secure workflows and enhance the general resilience of terraform infrastructure deployments.

1.7.2. Research Domain Contribution (Technological Contribution)

This work provides a new technical paradigm by marrying cryptographic provenance designs (Sigstore, Cosign, in-toto) with Terraform's supply chain. Others have investigated misconfiguration analysis and CI/CD security, but to the knowledge we are the first to apply SLSA style provenance and in-toto attestations to infrastructure definition code. The research will contribute to the literature by demonstrating how attestation based verification can be utilized to secure cloud configurations and elements. The prototype implementation and empirical analysis will yield the first evidence based insights into practicability, performance, output noise, and limitations of provenance enforcement in Terraform.

1.8. Research Challenges

This work offers some technical and practical challenges that need to be studied in the order to conduct this research successfully. Tool Integration Terraform was not built with cryptographic signing or verification in mind. This will be potentially complex to implement into their plugin model and command line interface, This could also involve changing or extending internal components.

Key Management: Operating cryptographic keys for signing modules and plans is operationally complex, particularly in the case of collaboration. The tension is in providing strong security controls such as those around key storage, rotation and revocation while maintaining the friendliness that developers need.

Governance overhead: Signing and verification can incur notable latency on CI/CD pipelines. Acceptable performance benchmarks and tuning of cryptographic operations to induce as low runtime overhead as possible will be needed.

End to End Cryptographic Checking: Present-day systems, including Git commit signature and provider signature are confined to standalone services. There is no cryptographic verification on a single run across the pipeline which ensures tamper evidence of Terraform plans, modules, and deployment states. The proposed research should thus form an unbroken chain of trust throughout all the lifecycle of the infrastructure artefacts.

Policy Enforcement and validation: In plan validation, tools such as the Open Policy Agent (OPA) and hashicorp sentinel are used to conduct point in time policy checks. Nevertheless, no ongoing cryptographic demonstration of compliance is supplied by them. The purpose is to come up with a mechanism that would guarantee sustained and verifiable implementation of security policies during the pre deployment and post-deployment stages.

Drift Detection and human Runtime Monitoring: The machine will alert when its goals are violated, and the operator will have the option to either adjust the goal or pause the machine,
The machine will notify the operator when its objectives are not met, and the operator will be given a chance to change the goal or stop the machine. Tools that are in place, like Driftctl and Spacelift, are based on reactive detection of drift. These methods have neither provenance nor tampered attribution. The study will be required to support proactive drift detection through cryptographically bound monitoring, and thus, avert real time infrastructure changes that are unauthorized.

Supply Chain Provenance: Whereas provider signing and provider static analysis (e.g., Checkov, tfsec) are used to check individual players, these tools do not offer supply chain protection. It is important to build a verifiable provenance chain connecting all Terraform modules, plans, and states and make them trustworthy all the way to the source code to the deployment. Digital Compliance and Digital Forensics: The existing Terraform auditing is based on logs and manual snapshots, which can be changed or not complete. The study will realize the use of the immutable,

cryptographically time-stamped audit records, which will give violation, incident response, and governance evidence of the compliance with the forensic standard. Assurance of Business

Continuity and Trust: Constant checkups on verifications and disaster recovery implementation currently relies on manual checks of compliance. By taking a cryptographic proving of compliance into automated recovery mechanisms, resilience and business continuity will be increased, so that even in times of disruptions infrastructure will be provably secure.

Usability and Adoption: Developers are reluctant to use new security workflows that do not fit with current security practices. Making the proposed solution jive well with Terraform's flow, whilst still being intuitive, will take careful consideration on design (and probably documentation, maybe even some opt in modes).

1.9. Research Questions

Considering the motivations, gaps in literature, and the main objectives of this research study, we rely on the following guiding questions:

1. How can cryptographic signatures, attestations, and transparency logs be used from top to bottom through the Terraform workow to offer end to end verication without sacricing developer productivity?
2. Which combination of signing schemes (e.g., Cosign signatures, in-toto attestations, TLog entries) provides the best effective protection against Terraform supply chain attacks?
3. Key Management: What can be done to simplify key management through cryptographic keys to enable adoption within the developer space; e.g., auto issuance of certificates or short lived signing keys?
4. What is an acceptable performance overhead of the signing and verification in Terraform CI/CD pipelines, and how does it compare to baseline workflows without provenance checks?

These research questions will inform the development, deployment, and evaluation of the cryptographic provenance for Terraform. They become the basis of this study’s contribution to IaC security and DevSecOps research more generally. Verdet *et al.* (2023)

1.10. Research Aim

The aim of this project is to design, implement, and evaluate a cryptographic provenance framework for Terraform IaC. The core idea is to leverage digital signatures and immutable references so that every module and plan in a Terraform workflow can be verified end to end.

Specific objectives include:

- Threat analysis: Assess the threat model and attack vectors relevant to Terraform supply chains (building on analyses by BoostSecurity and others) (Proulx, 2023; Schneeweisz, 2022).
- Prototype development: Develop a prototype system that integrates code signing (e.g. using Sigstore/Cosign) into the Terraform workflow, automatically producing signed provenance metadata for modules and deployments.
- Provenance integration: Incorporate SLSA and in-toto principles to create verifiable build provenance for Terraform configurations. For example, implement in-toto "link" files that record module hashes and have them signed.
- Evaluation: Test the solution's effectiveness in real-world IaC scenarios. This includes intentionally injecting malicious changes into open-source Terraform modules to verify that this system detects them, as well as measuring performance impact.
- Recommendations: Document recommendations for practitioners on adopting cryptographic provenance in DevSecOps pipelines. This may include proposals for new Terraform features or best practices informed by this project.

These objectives directly target the trust gap highlighted by HashiCorp and others (HashiCorp, 2024), and achieving them will fulfill the project's aim.

1.11. Research Objectives

Achieving these objectives will answer the research questions and meet the expected learning outcomes, such as understanding security in DevOps, conducting a thorough literature survey, performing requirements analysis, designing and implementing a security system, and evaluating it systematically.

ID	Description	Justification	Learning Outcomes (LOs) Mapped	Research Questions (RQ) Mapped
R01	Review existing Infrastructure as Code (IaC) security practices and provenance frameworks.	Conduct an in depth literature review on Terraform security, in-toto, Sigstore, and SLSA frameworks. This ensures a strong understanding of existing methods and informs the system design phase.	L01, L04	RQ1, RQ2
R02	Design and develop a prototype framework for signing and verifying Terraform modules and configurations.	Implement integration with Terraform (e.g., plugin or wrapper) using cryptographic signatures and attestations. This provides the core technical contribution to test the feasibility of provenance based IaC security.	L01, L02	RQ1, RQ2
R03	Evaluate the prototype under simulated supply chain attack scenarios.	Test the system using tampered modules and version hijacks to measure detection	L01, L03	RQ3, RQ4

		accuracy, false positives, and performance overhead. This validates system effectiveness and efficiency.		
R04	Analyze usability and developer adoption factors.	Gather feedback from DevOps practitioners through surveys or interviews to assess ease of use and workflow impact. This ensures the solution is practical and user friendly.	L03, L04	RQ3, RQ4
R05	Compare the proposed provenance framework with existing Terraform security practices.	Benchmark the approach against current methods such as Git hash pinning and manual verification, to quantify improvements in integrity and trust.	L01, L04	RQ2, RQ4

Table 2: Research Objectives

1.12. Hardware/Software Requirements

Category	Item / Tool	Specification / Version	Purpose / Usage
Hardware	Development Machine (Laptop/Server)	Intel Core i7 / AMD Ryzen 7 (8 cores, 16 threads), 16 GB RAM, 512 GB SSD	Primary development and testing environment for Terraform IaC workflows and cryptographic signing integration.
	Optional GPU (for simulation or CI/CD containerization)	NVIDIA RTX 3060 or equivalent, 6 GB VRAM	For running containerized workloads or performance simulations if needed. Not used for model training, but for testing signing and verification scalability.
	Network Environment	Stable internet connection (≥20 Mbps)	Required for connecting to Terraform Cloud, GitHub, Sigstore Fulcio/Rekor APIs, and CI/CD pipelines.
Software	Operating System	Ubuntu 22.04 LTS / Windows 11	Base OS for Terraform development and testing environment.
	Terraform CLI	v1.7.x (HashiCorp)	Infrastructure as Code tool for provisioning cloud environments and integrating cryptographic provenance.
	Terraform Cloud	Latest version (Managed SaaS)	Remote state management and secure collaboration for Terraform projects.
	Sigstore Cosign	v2.x	For signing and verifying Terraform modules, plans, and provenance metadata.

	Sigstore Fulcio	Managed service	Issues short-lived signing certificates for CI identities (keyless signing).
	Sigstore Rekor	Managed service	Transparency log for storing and auditing Terraform artifact signatures.
	in-toto Framework	v1.3+	Generates and verifies provenance attestations for Terraform pipeline steps.
	Python	3.11	Scripting for test automation, provenance verification scripts, and evaluation metrics.
	Bash / Shell Scripts	GNU Bash 5.x	Automation scripts for Terraform workflows and CI/CD integrations.
	Git and GitHub	Git 2.44+, GitHub (cloud)	Version control, repository hosting, and CI/CD automation via GitHub Actions.
	GitHub Actions	Latest	CI/CD automation platform for Terraform workflow execution, signing, and verification.
	VS Code (Visual Studio Code)	Latest	Integrated development environment for writing Terraform configurations, Python scripts, and CI/CD workflows.
	ClickUp	Cloud platform	Project management and task tracking for research planning and progress monitoring.

	Syft (Optional)	v1.x	Generates Software Bill of Materials (SBOM) for Terraform pipeline transparency.
	Docker (Optional)	v25.x	For containerizing CI/CD environments and reproducible Terraform pipeline testing.
	Google Docs / Microsoft Word	Cloud / Office 365	For writing project documentation and final report.

Table 3: Table of Software and Hardware Requirements

1.13. Chapter Summary

In this chapter, we introduced the importance of securing Terraform IaC through cryptographic provenance. Here it has defined the core problem and explained it with recent studies and real incidents and summarized existing approaches (and their shortcomings) and identified a clear gap in Terraform supply chain security. Then this has outlined the contributions, challenges, research questions, and objectives of the project. This sets the stage for the next chapter, which will review related work in detail and provide a foundation for the methodology.

CHAPTER 02: LITERATURE REVIEW

2.1. Chapter Overview

The problem domain this chapter reviews is securing infrastructure as code, with respect to Terraform and cryptographic provenance. In here introduce important concepts (IaC basics, supply chain security, provenance). Afterwards, scrutinize any existing material, such as known practices in Terraform security, security frameworks in supply chains (such as Sigstore and in-toto), and any previous attempts to add attestations to IaC. And also cover datasets and tools that are applicable to this study and describe this evaluation and benchmarking of existing solutions. Lastly, conclude the major findings of the review and show how it informs practice.

2.2. Problem Domain

A basic DevOps practice, Infrastructure as Code (IaC) is an automation method used to deploy, manage, and configure resources in the cloud. Declaratively defining infrastructure (such as in the language of Terraform HCL) allows teams to version manage their environments and can be reliably instantiated (Lepiller et al., 2021). In IaC, quality and security management play a vital role since misconfigurations might result in severe outage or breaches. Programs such as Checkov, TFLint can be used to analyze Terraform code to help identify bugs and security concerns. To give an example, a survey of common defect patterns in IaC scripts (hard coded credentials or open ports) conducted by Verdet et al. (2023) indicated that the encryption at rest policy is one of the least considered within projects. It means that some security measures (such as appropriate access controls) are used, but there exist a lot of valuable security practices that are not widely implemented. It implies that critical checks have to be automatically enforced, possibly via cryptography.

The IaC domain deals with the notions of modules (code packages that can be reused) and state files (maintaining deployed resources). Threats such as code injection, secret leaking, and supply chain tampering should be considered in securing IaC. The main concerns are the integrity of the code, safe processing of sensitive variables, and the safety of the execution environment. Due to the ability to distribute module sources (the Terraform Registry, Git repositories, and so on), trust

needs to be handled explicitly. In contrast to traditional software applications, IaC code frequently has direct access to running infrastructure, allowing a vulnerability or malicious modification in IaC to have direct and immediate widespread effects.

Terraform being one of the most popular IaC tools, comes with some distinct implications related to its declarative model, large provider ecosystem, and the use of state locking mechanisms. It fits inside a complicated software supply chain consisting of public/private module registries, version control systems, CI/CD pipelines, and artifact repositories. A potential attack surface is each of the links in this chain. Programs such as Supply chain Levels for Software Artifacts (SLSA) are a general approach to enhancing supply chain security, and Terraform specific implementations remain scarce. The incorporation of security into DevOps (DevSecOps) is also a factor: it demands the ability to balance between automation and control. Cryptographic provenance mechanisms should be implemented in a way that facilitates, and not inhibits, developer workflows. That is to say, to offer security without too much friction.

Moreover, Terraform is at the heart of cloud security posture management as it determines the desired state of the infrastructure. Before deployment, it can impose some security policies (such as policy as code inspections using tools such as Sentinel or Open Policy Agent). Nevertheless, the current advice (e.g. the CI/CD "cheat sheets on IaC created by OWASP) is not concerned with module authenticity, but rather with the limitation of analysis and manual gating. This is the place, where the cryptographic provenance might provide a new defense, that the code deployed is exactly the one it has to be.

2.3. Existing Work

Citation (Year)	**Methodology**	**Contributions**	**Limitations**	**Relevance**
Verdet *et al.* (2023)	Empirical study of Terraform projects (n=812)	Taxonomy of IaC security vulnerability patterns	Focus on identification; no mitigation proposed	Foundation for the threat modeling

Palma *et al.* (2024)	Static analysis tool for Terraform security	Automated detection of IaC misconfigurations	Does not address runtime or supply chain issues	Complementary policy enforcement
Xu *et al.* (2025)	Empirical study of IaC dependency updates	Insights into module dependency management	Limited direct security recommendations	Inform module verification design
Anderson (2023)	Analysis of Sigstore public instance	Keyless signing patterns, use cases	General; not Terraform-specific	Foundation for signing approach
OWASP (2023)	Community survey (CI/CD risks)	Comprehensive CI/CD risk taxonomy	Not IaC-specific	Risk assessment framework

Table 4: Existing Work with Relevance

2.3.1 IaC Security Practices

The best practices around Terraform and other IaC have been studied. Verdet et al. (2023) discovered that most basic policies (such as IAM least privilege) are implemented, but other best practices (such as encryption of data) are often overlooked. Before deployment, a certain number of misconfigurations can be identified by a set of tools (e.g. Checkov, TFSec) relying on a static analysis of the code but do not ensure its authorship or integrity. Things such as secrets management and least privilege are highlighted in community best-practice guides (such as OWASPs IaC cheat sheets), but generally, do not cover the topic of verifying module authenticity. Guidelines on the industry (e.g. the Terraform security recommendations by Sysdig) recommend not to use secrets in code and secure state backends, yet they do not propose cryptographic verification of code provenance (Douglas, 2023).

In general, the current research on IaC security has focused on identifying insecure code or infrastructure patterns (such as hardcoded secrets or overly liberal permissions). To illustrate,

Verdet et al. chronicled typical problems (lack of encryption, etc.) in AWS, Azure, and GCP Terraform projects. But dependencies (modules/providers) on the supply chain side induce a weakest link effect: it is only secure code on your part that can prevent a dependency that is compromised. This is not new in general software supply chain research. An IaC environment can be characterized by each execution of terraform apply potentially bringing in new modules, an unwanted change in a module can be very dangerous. Large scale incidents such as SolarWinds or Codecov demonstrate this threat: by adding malicious code into the build or delivery processes, it is possible to compromise thousands of deployments. As mentioned, the OWASP Top 10 CI/CD Risks list encompasses Dependency Chain Abuse and Pipeline Poisoning, with the authors stressing that attackers would use the seamless means of external dependencies retrieval to implement malicious code (OWASP, 2023). These observations indicate that IaC security requires not just code checks but also supply chain verification processes.

2.3.2 Supply Chain Security for Terraform

There have been recent debates on the vulnerabilities of the supply chain in Terraform. As shown by Schneeweisz (2022) (GitLab blog), Terraform modules do not have cryptographic signatures, and can be compromised through tampering with the Git history. He demonstrated attacks in which malicious commits were introduced in the place of the mutable Git tags. His proposed mitigations were pinning to full commit hashes and validating Terraform lockfiles, which are manual processes and not enforced. In 2022, HashiCorp recognized the risks of modules; a forum post by them, for example, states that they should improve the security of the registry and recommends caution when using module sources. A key limitation identified by research conducted by BoostSecurity in 2023 was the use of Terraform registry modules not locked by content hash. In contrast to providers (which Terraform downloads as versioned binaries and verifies with the lockfile SHA-256), modules were specified by a version tag (Proulx, 2023). This implied that Terraform did not provide any cryptographic assurance that a program can be executed with the identical module code each time.

According to the BoostSecurity report, there is no assurance that a plan created a few seconds after that would act in a similar manner (Proulx, 2023) as the code of a given module may alter. They also found numerous vulnerable CI configurations that could be hijacked by malicious pull

requests. As a result of these findings, HashiCorp published an official bulletin (HCSEC-2024-04). As of January 2024, the public Terraform Registry has adopted the behavior of versioning modules using immutable Git commit SHAs and fixed repository IDs (HashiCorp, 2024). This makes sure that the published code of a module version can never be modified after the fact the registry will provide the commit that was initially uploaded. Although this is a move towards provenance, it is not a mandatory feature (users can still access modules external to the registry) and does not apply to on premises or modules managed by CI. Hence, Terraform ecosystem does not have a provenance solution that would be universal to all modules, which leads to the desire of the framework.

2.3.3 Cryptographic Provenance and Attestations

Beyond the Terraform scenario, software supply chain security research has been actively developed. Sigstore is a cryptographic signing and software supply chain transparency toolkit project of the OpenSSF. Cosign (a Signatory tool) can sign container images or any file, and Rekor is an append only public log of signatures (Sigstore, 2023). A framework, in-toto (originally an academia project but currently a CNCF project) defines a process in which every stage of a software process results in a signed link file, providing an end to end provenanced record of building (Sirish and Kennedy, 2023). In-toto has, via its implementation in compiled software and containers, been widely used, but it is general and can be applied to IaC processes. As one example, in-toto may be used to sign metadata at each of the steps in a Terraform deployment (fetch modules, plan, apply). Earlier scholarly literature on host based integrity (e.g. Stefan et al., 2009) demonstrated the promise of cryptographically authenticating system inputs, yet more recent industry commentary discusses how modern provenance tools effectively render tampering mathematically infeasible (Igboanugo, 2025). Standards such as SLSA are used to describe degrees of confidence in build pipelines. SLSA focuses on the production of signed provenance statements to demonstrate the construction of artifacts according to expectation.

As stated, Tamanna et al. (2024) discovered that the risk of supply chain attacks may be reduced substantially by using such practices as: source integrity verification and provenance tracking. These practices are starting to become common in the industry: a notable example is the addition of Sigstore in the npm package registry in 2023, which subsequently results in a signed provenance

attestation being sent with every package release (Sigstore, 2023). Similarly, the DevSecOps best practices promoted by Microsoft recommend that all items passing through CI/CD be digitally signed (Microsoft, 2022). These tendencies mean that cryptographic provenance is emerging as a software security expectation. None of these concepts, to the best of knowledge, have been specific to the deployment of Terraform and IaC in the published literature. The design of HashiCorp does not offer a complete secure supply chain (the provider lockfile) yet. No similar system exists to check the integrity of modules board wide (Proulx, 2023). The efforts will help fill this gap by implementing established supply chain security methods to Terraform.

2.3.4 Related Technologies

Several technologies related to the research deserve mention,

- Sigstore & Cosign: As noted, Cosign can sign and verify container images and arbitrary files. Explore using Cosign to sign Terraform module archives (for example, a `.zip` of the module directory). The signed records can be stored in Sigstore's Rekor log for transparency (Sigstore, 2023).
- in-toto: In-toto allows defining a supply chain layout (steps and expected materials/products) and then generating signed link metadata for each step. Might define a layout where terraform init and terraform apply are steps. During init, an in-toto link could record the hashes of all downloaded modules; during apply, another link could record the hash of the resulting state file. Each link would be signed by the CI system or developer performing the action (Sirish & Kennedy, 2023). This would create a chain of trust that the infrastructure was applied exactly as planned, with no tampering in between.
- Terraform Lockfile: Terraform's existing lockfile secures provider binaries by locking their checksums. This could incorporate provider checksums from this lockfile into overall provenance record (to ensure providers were not tampered with). However, as noted by prior work, the lockfile must itself be verified to be effective. User the lockfile as an input to the signature process (e.g. sign the lockfile or include its content in attestations).
- SBOMs: A Software Bill of Materials could be generated for a Terraform configuration listing all modules and providers (and possibly even the specific versions of each cloud resource type). While an SBOM by itself cannot prevent tampering, a signed SBOM could

be used at verification time to ensure that the modules used match the expected list. There are tools like Syft that can create SBOMs for IaC, which might use in combination with signing (though this is optional).

2.3.5 Critique of Existing Approaches

From the review above, can see that current IaC security measures are largely non cryptographic. Manual or procedural controls (like code review and version control) and static analysis are helpful but can fail against a determined adversary who exploits the supply chain. The state of the art supply chain security tools (Sigstore, in-toto, etc.) have shown their value in other domains (containers, software builds) but have not been integrated into Terraform workflows. To the knowledge, no published solution explicitly signs Terraform configurations or modules and then verifies them during deployment.

Some new tools and studies are worth noting as partial steps. TerrARA (Tran *et al.*, 2025) is an automated threat modeling tool for Terraform. It generates abstract data flow diagrams from Terraform configurations and identifies potential security threats in the design. TerrARA can highlight vulnerabilities in the infrastructure design (like overly open network paths), but it assumes the code itself is trusted and static. It does not address dynamic supply chain tampering or verify the provenance of the code being modeled. Thus, even advanced tools like TerrARA do not solve specific problem.

In conclusion, although cryptographic guarantees have been studied and implemented in general software supply chain contexts (such as container image signing and build attestations), there is a clear gap in applying these guarantees to Terraform's infrastructure pipelines. This proposal aims to fill that gap by leveraging lessons from those related domains while focusing on the unique challenges of Terraform. The literature review justifies the need for this project and has informed the design of this methodology (outlined in Chapter 3).

2.4. Dataset Selection

To assess it, will consider publicly accessible Terraform modules and code repositories. There are thousands of AWS, Azure, Kubernetes, and other modules in official Terraform Registry and

GitHub. The modules that will use to test the system are representative (as is common in activities such as VPC networking, IAM roles, or configuration of a Kubernetes cluster) and will be selected randomly. Can also produce fake versions of some of the modules that are tampered (e.g. injecting a resource that exposes a security hole). Moreover, take advantage of any open IaC datasets on the market (e.g. the multi-cloud Terraform configuration dataset that was examined by Verdet et al. 2023).

The preparation of the data will mean extracting or wrapping module source code in a way that will be signed. As an illustration, we might have to programmatically download modules, zip up their contents (to form a fixed artifact to sign) and create any required lockfiles or metadata, also capture the appropriate metadata such as the name of the module, version and the source location to be utilized in our evaluation.

2.5. Benchmarking and Evaluation

In order to assess the cryptographic provenance system, measure several parameters of its functionality and security:

- Security Effectiveness: Put the system through a known attack vector dataset using the tampered module dataset. The critical measurements will consist of the rate of detection (true positives), false positives, and the rate of prevention of each category of attacks. To give an example, should our system be able to reliably track and reject an altered module that has been signed after it was signed? Will emulate such scenarios as code injection into modules and manipulation of version tags, as well as use of other binaries without permission to make sure that our methodology can detect them.
- Performance Impact: Will quantify the overhead that the signing and verification steps will introduce. The most important metrics are the time to sign a module, time to check a module, and the total effect on the pipeline execution time. Scalability will also be considered the way that the overhead increases with the number or size of modules. These figures will be checked against the results of deployment at the time of no signing. Ensure that performance is reasonably maintained (e.g. adding only a few seconds to the standard run of a pipeline).

- Usability Evaluation: The developer experience will be evaluated by user study or expert opinion. This could entail a small team of DevOps practitioners using our tool-enabled to do a Terraform deployment task as opposed to the normal workflow. These measures will be time of task completion, rate of errors (e.g. whether the signing process is confusing or introduces mistakes) and subjective data on whether the process is easily usable and whether it introduces friction. This will assist in the feasibility of our solution implementation in actual teams.

Below table outlines key evaluation metrics, the current baseline (existing approaches), and our target performance goals,

Metric	**Existing Approaches**	**Target Performance**	**Measurement Method**
Module Verification Time	N/A (not implemented)	< 1.5 seconds per module	Benchmark testing on sample modules
Plan Signing Overhead	N/A (not implemented)	< 2 seconds per plan	Performance profiling on pipelines
False Positive Rate	N/A (not implemented)	< 3% for policy checks	Attack simulation testing
Detection Rate	N/A (not implemented)	> 98% for known attacks	Attack simulation testing

Table 5: Evaluation Metrics Comparison

As shown, existing Terraform processes do not implement these verification steps, so there is no baseline measurement (N/A). Our framework aims to introduce the steps with minimal overhead (seconds) and high security (high detection, low false alarms). Will verify these targets through the methods described.

2.6. Chapter Summary

This literature review highlighted that while various security practices exist for Terraform (static analysis, policy as code, etc.), none currently ensure full code provenance or prevent supply chain tampering. Cryptographic provenance methods (like Sigstore signing, in-toto attestations, and

SLSA guidelines) show promise in other domains but have not yet been applied to IaC. Can identified key gaps, notably the lack of module signatures and automated verification in Terraform workflows, outlined relevant metrics for evaluating our work (such as detection rate, overhead, and usability). In summary, there is a strong justification for the project. The insights from this review have directly informed the design of our methodology which can present in the next chapter.

CHAPTER 03: METHODOLOGY

3.1 Chapter Overview

This chapter describes the research and development methodology for the project on securing Terraform IaC with cryptographic provenance. First outline the overall research strategy (following a pragmatic approach, akin to Saunders' "research onion" model). Next, detail the development methodology, including how can gather requirements, design the system, and perform testing. Then explain the planned implementation steps for our solution (integrating signing, verification, and pipeline changes). Finally, cover project management aspects such as scope, schedule, and risk mitigation. This chapter establishes how can conduct the research systematically and ensure rigor in both building the solution and evaluating it.

3.2 Research Methodology

Layer	**Choice (What is Being Used)**	**Justification (Why You Are Using It)**
Philosophy	Pragmatism	Combines theory (security principles like SLSA, in-toto) with practical validation (Terraform prototype). Focuses on what works effectively in DevSecOps environments.
Approach	Deductive	Applies existing security theories (e.g., provenance and attestation frameworks) to Terraform, then tests them

		empirically to validate feasibility and performance.
Methodological Choice	Mixed-Method (Quantitative + Qualitative)	Quantitative testing for performance and accuracy; qualitative assessment via expert feedback on usability and workflow impact.
Strategy	Experimental / Design Science	Involves building a prototype artifact (signing-enabled Terraform workflow) and testing it against controlled tampering and attack simulations.
Time Horizon	Cross-Sectional	The study will be conducted within a fixed timeframe to evaluate a single version of the prototype, rather than over an extended longitudinal period.
Techniques and Procedures	Engineering Development + Static/Dynamic Analysis	Development involves integrating Sigstore/Cosign and in-toto within Terraform pipelines. Evaluation includes simulated attacks, signing/verification benchmarking, and expert feedback collection.

Table 6: The layers of Saunders' Research Onion

3.3 Development Methodology

3.3.1 Requirements Analysis

Collect functional requirements and security requirements of the system. This includes finding the most crucial trust limits in Terraform processes, deciding what cryptographic keys or identities are required, and learning about real world limits of DevOps teams

Threat Modeling: Review the Terraform pipeline that is already in place to identify the areas where it can be tampered with (e.g. fetching modules, during plan/apply, storing state). Determine what attacks must defend (such as substituting module code with malicious code, compromised provider binaries, etc.).

Interviews/Surveys: Converse with a minimum of five DevOps engineers or users of Terraform to understand how they currently handle the issue of managing dependencies in Terraform and their pain points or needs regarding security. Going to enquire about how they would feel about signing modules, what kind of an overhead would be acceptable, and what keys would be handled.

Literature / Advisory Review: Include advisories provided by authors such as OWASP CI/CD risk list, HashiCorp security advisory and postmortem of incident case. These will aid in making sure that take into consideration all the plausible threat scenarios. Out of such activities, we will obtain several requirements. As an example, one of the requirements can be expressed as the system will be able to confirm the integrity of every Terraform module before apply or the signing process will not require more than 2 seconds per module to the pipeline. Through the creation of such requirements, can develop the system to satisfy them and subsequently check whether they are satisfied.

3.3.2 System Design

Based on the obtained needs, a modular signing and verification structure in Terraform will be developed. The main elements of the design can be: Signing Service: A sub procedure (possibly a script but could also be a CI job) which signs with Sigstore/Cosign module archives or Terraform plan files. It will connect to a key management system (either via a local private key or by acquiring a certificate by Sigstore's Fulcio).

Verification Hook: A kind of hook into the Terraform apply process (perhaps through a wrapper script or as a Terraform provider plug-in) that verifies some signatures before deployment continues. This could either be in the form of verifying a signature by parsing a signed metadata file or by accessing the Rekor transparency log.

Layouts/Attestations: Can state an in-toto layout of how the Terraform pipeline is supposed to be run. E.g. it could state that following a terraform init, an attestation of all module hashes should be generated, and that it is signed by the CI service. After that, prior to terraform apply, the system should ensure that attestation. The input/output (state file, modules) of each step could thus be checked.

Transparency Log Interface: Allow connection to Rekor (or whatever other log) such that all the signature values are also logged in a public log. This gives an additional tamper-evidence and auditing capability (somebody can later query the log to verify the presence and integrity of a signature).

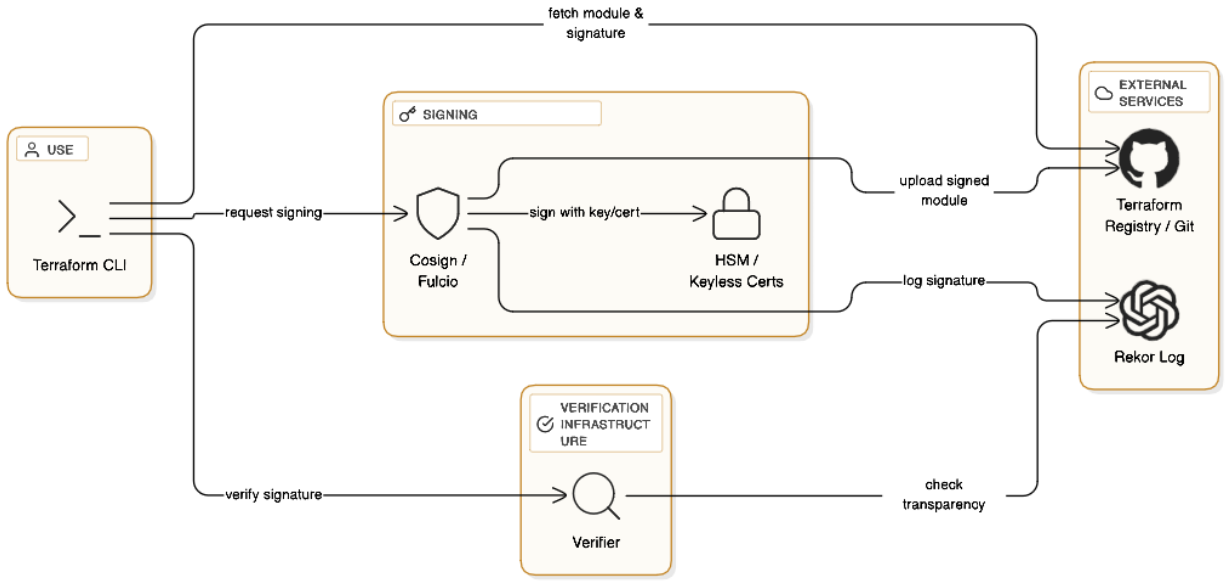


Figure 2: Architecture Diagram (Self-composed)

The design will emphasize backward compatibility and minimal intrusion. For instance, signing could be an optional step that, if skipped or if verification fails, will block deployment (fail safe). We will prepare UML or architecture diagrams to illustrate how data flows through our system, where trust is anchored, and how threats are mitigated at each point.

3.3.3 Implementation Steps

The prototype will be deployed in stages, as follows,

Setting Environment: prepare the development and testing environment. This may include installing a CI pipeline (e.g. GitHub Actions or GitLab CI) and a sample project using Terraform to use. And also set up services required (such as a local rekor server where needed, or access to the public Sigstore services).

Cosign Integration: Application of Cosign module signing. As an example, once terraform init downloads modules, might have a script which packages the files of each module and calculates a signature of it. Alternatively, can pass the whole terraform module directory as a large artifact. Signature storage Signatures can be stored anywhere and in any manner. An immutable record will

be done using Rekor, but the signature data must be available to the deployment process as well. This may be either through a signature file which is kept with the modules (e.g. a file such as module.tar.sig) or by an API call to Rekor at the time of verification.

Verification Hook: Build a Terraform CLI wrapper or a plugin, which manages to intercept the apply step.This hook will ensure that before changes are being applied, the content of each module is known to be a good signature. In case of any verification failure, then it will roll back the deployment. This element may refer to the local signature files or Rekor to validate authenticity.
In-toto Attestation (extension): Under time constraints, in-toto attestation of the pipeline is optional. This means that as an example, create an in-toto link file after the planning phase which captures what was planned to deploy and another after apply capturing what was deployed. Both link files would be signed. This would provide complete provenance trace of the deployment process.

User Tooling: Put these steps together into an easy to use tool or script. It might be a wrapper script (i.e. Python or Go program that developers execute rather than run Terraform directly) or a Terraform Cloud/Enterprise integration. It can also be offered as a GitHub Action or other CI plug-in to make it easy to use.

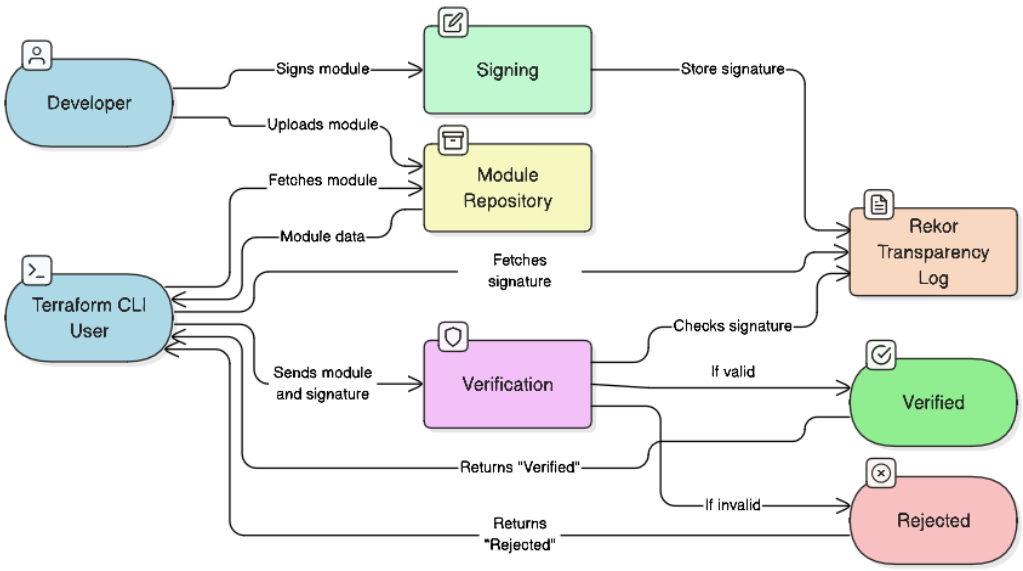


Figure 3: Component Diagram (Self-composed)

At each step, document the decisions, test incrementally, and adjust based on feedback. For key management, as noted, and may utilize Sigstore's Fulcio to get on-demand code signing certificates tied to developer identity (using OIDC). This avoids managing long term private keys manually and can improve usability (developers wouldn't need to distribute keys, just authenticate to GitHub, for example).

3.3.4 Testing and Evaluation

Security Tests

Develop situations that the system is expected to identify problems.

For instance,

(a) Edit an already signed module repository (inject a malicious resource).

(b) Edit a version tag or release a different module with dissimilar content without re-signing.

(c) Effort to make use of an unsigned module when signatures are on.

Operate the pipeline in such situations and ensure that it rejects the manipulated inputs. True positive rates (effective detectives of actual tampering) and false positive rates (detected as such when it should not have been) will be measured.

Performance Tests: Compare the time taken to deploy Terraform (without signing) with the time taken to deploy Terraform with the provenance checks. Measures comprised time to sign one module, time to verify a module and the time CI pipeline used in a typical project. To check the scalability of the system will test with small projects (a few resources), medium sized projects (dozens of resources and several modules), and large projects (hundreds of resources across modules).

Usability Tests: Ask few users (colleagues, or volunteer DevOps engineers) to use the tool. A typical Terraform deployment assignment may be offered to them and the comparison with the usual process is followed. Will mention whether they face any confusion or mistake and collect their response regarding how comfortable they would be in implementing it in practice. The metrics might involve time and error rate and completing tasks, and qualitative feed. The results of all tests would be recorded and processed.

3.4 Project Management

3.4.1 Scope and Deliverables

The project scope includes developing and evaluating a working prototype of the system, but it does not include deploying it to production environments at scale. Key deliverables will be,

1. Literature Review & Requirements Document: A report summarizing the background research and listing the specific system requirements derived from it.
2. Design Artifacts: Architecture and design diagrams, threat model documentation, and an explanation of how the design meets the requirements.
3. Prototype Code: A working implementation (likely hosted in a public Git repository) that includes the Terraform integration, signing/verification logic, and any auxiliary scripts or configurations.
4. Evaluation Report: Documentation of the test scenarios, results, and analysis for security effectiveness, performance, and usability.
5. Final Project Report: The complete write-up of the project including introduction, methodology, results, and recommendations for future work or practical adoption.

Out of scope for this academic project is fully productizing the solution (for example, covering every Terraform feature or performing a large-scale industry rollout). Also do not anticipate a large user study due to time constraints, focusing instead on a small-scale usability check. The intent is to demonstrate feasibility and benefits in a controlled setting.

3.4.2 Work Plan and Schedule

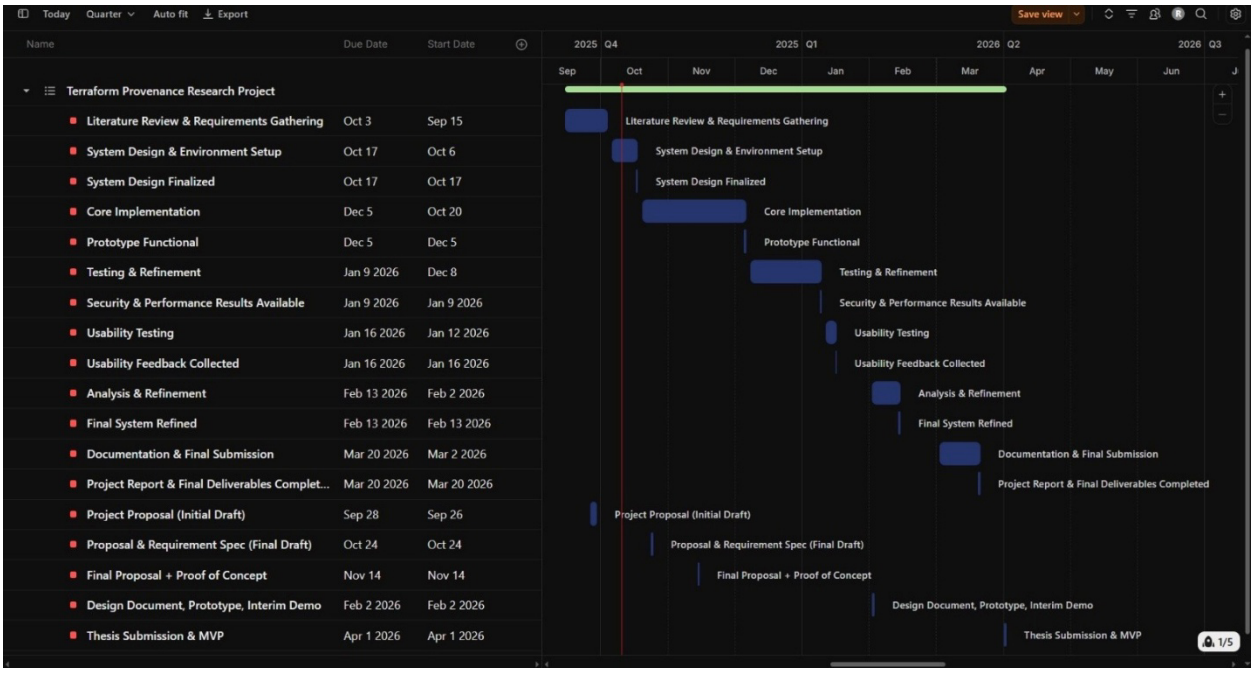


Figure 4: Work Plan (Self-composed)

3.4.3 Risk Analysis

Key risks and mitigation strategies are summarized below,

Risk	**Severity (1–5)**	**Probability (1–5)**	**Mitigation Strategy**
Difficulty integrating with Terraform	5	3	Use Terraform's plugin API or CLI hooks; consult Terraform docs.
Key management complexity	4	2	Use Sigstore Fulcio for automatic key issuance; restrict use to test environments initially.
Cryptographic errors/unexpected failures	3	2	Thoroughly test with valid modules; implement clear error handling and logging.

Performance bottleneck	4	3	Optimize verification (caching, parallelization); measure and tune performance early.
Developer resistance to workflow changes	3	4	Make signing optional via flags; provide documentation, examples, and training.
Terraform feature changes	2	2	Keep the system modular; monitor Terraform changelogs and update integration accordingly.

Table 7: Risk Analysis

By continuously monitoring these risks (via weekly reviews) and applying the above mitigations, aim to ensure the project stays on track.

3.5 Chapter Summary

This chapter explained how to do the research and development. Shall take a pragmatic method first to analyse requirements and threats after which construct a prototype and ultimately test it stringently. The system design section described how to integrate signing and verification into Terraform with the available tools (Sigstore, in-toto) and how the parts will be integrated. To test our security our testing strategy will involve simulating attacks, to test our performance overhead, and to test usability our testing strategy will involve soliciting user feedback. Also have outlined project management such as scope, (scope revolved around a prototype, rather than the entire product), timeline of the tasks and risk management plan. The subsequent steps involved in this planning phase are to implement the plan: deploy the prototype system and subsequently test its security and performance with regards to the criteria we have established.

CHAPTER 04: Software Requirement Specification (SRS)

4.1 Chapter Overview

This chapter details the software requirements for the proposed solution to secure Terraform Infrastructure-as-Code (IaC) pipelines using cryptographic provenance. We begin by identifying the key stakeholders and their viewpoints, focusing on DevOps engineers and security teams who will use or be impacted by the system. Next, we outline the methodologies used to elicit requirements, including literature review, interviews, surveys, and internal brainstorming/observation sessions. We then discuss the findings from each method: insights from existing research and industry best practices, feedback from practitioners, quantitative survey results, and observations of current Terraform workflows. These findings are summarized to derive a clear set of system needs. Following this, we describe the system context (in lieu of a visual context diagram) to explain how the proposed tool interacts with users and external services in the Terraform ecosystem. We also describe the use case diagram in text form, enumerating the primary actors and use cases of the system. Detailed use case descriptions are provided for each core scenario. Finally, we consolidate the requirements in a MoSCoW prioritization format, separated into functional and non-functional requirements. The chapter concludes with a summary, highlighting how the requirements were derived and setting the stage for the system design in subsequent chapters.

4.2 Rich Picture Diagram

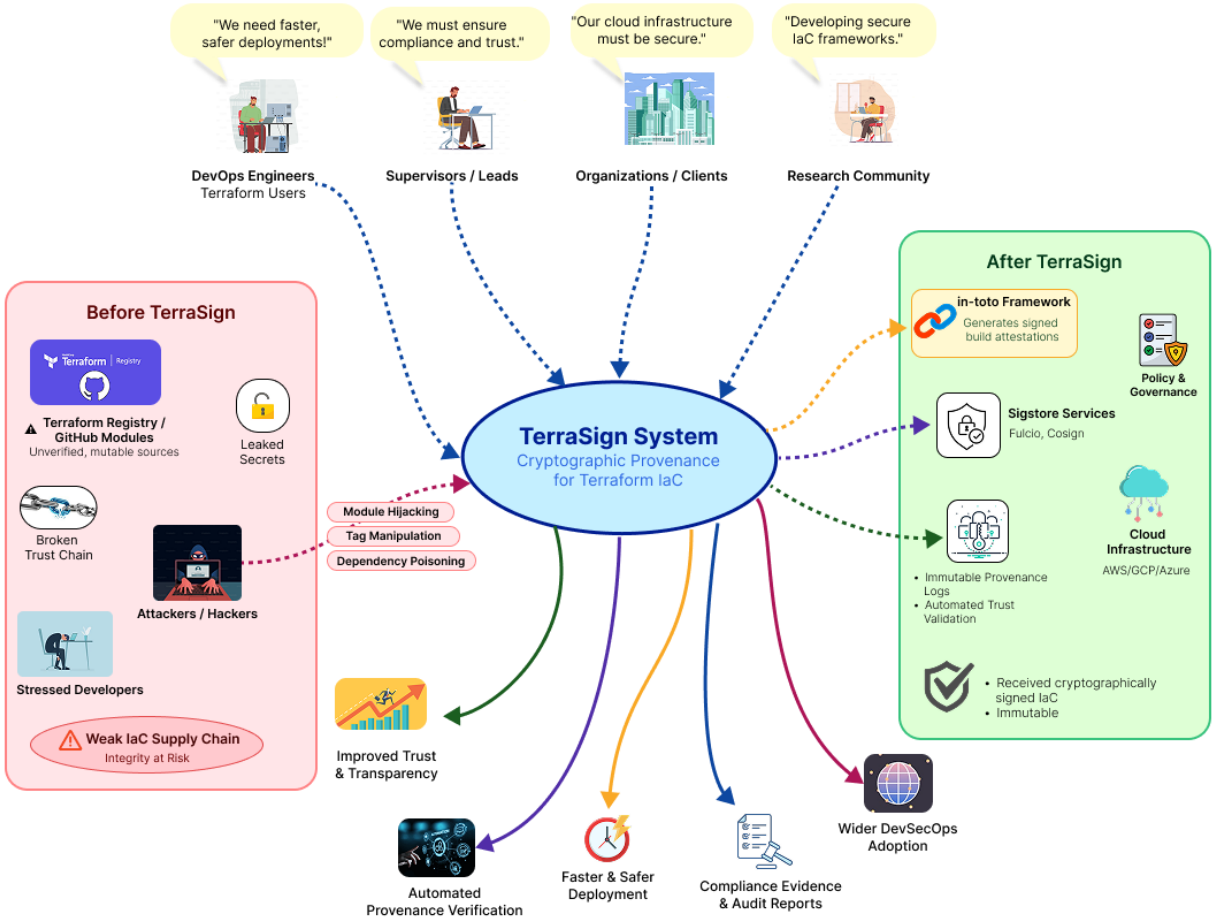


Figure 5: Rich Picture Diagram (Self-composed)

4.3 Stakeholder Analysis

4.3.1 Stakeholder Onion Model

The **Stakeholder Onion Model** presents a layered approach to identifying stakeholders based on their influence and involvement in the system. This model groups stakeholders into layers that surround the system and shows their relationships. It helps in understanding who has direct vs. indirect influence on the system and at what level.

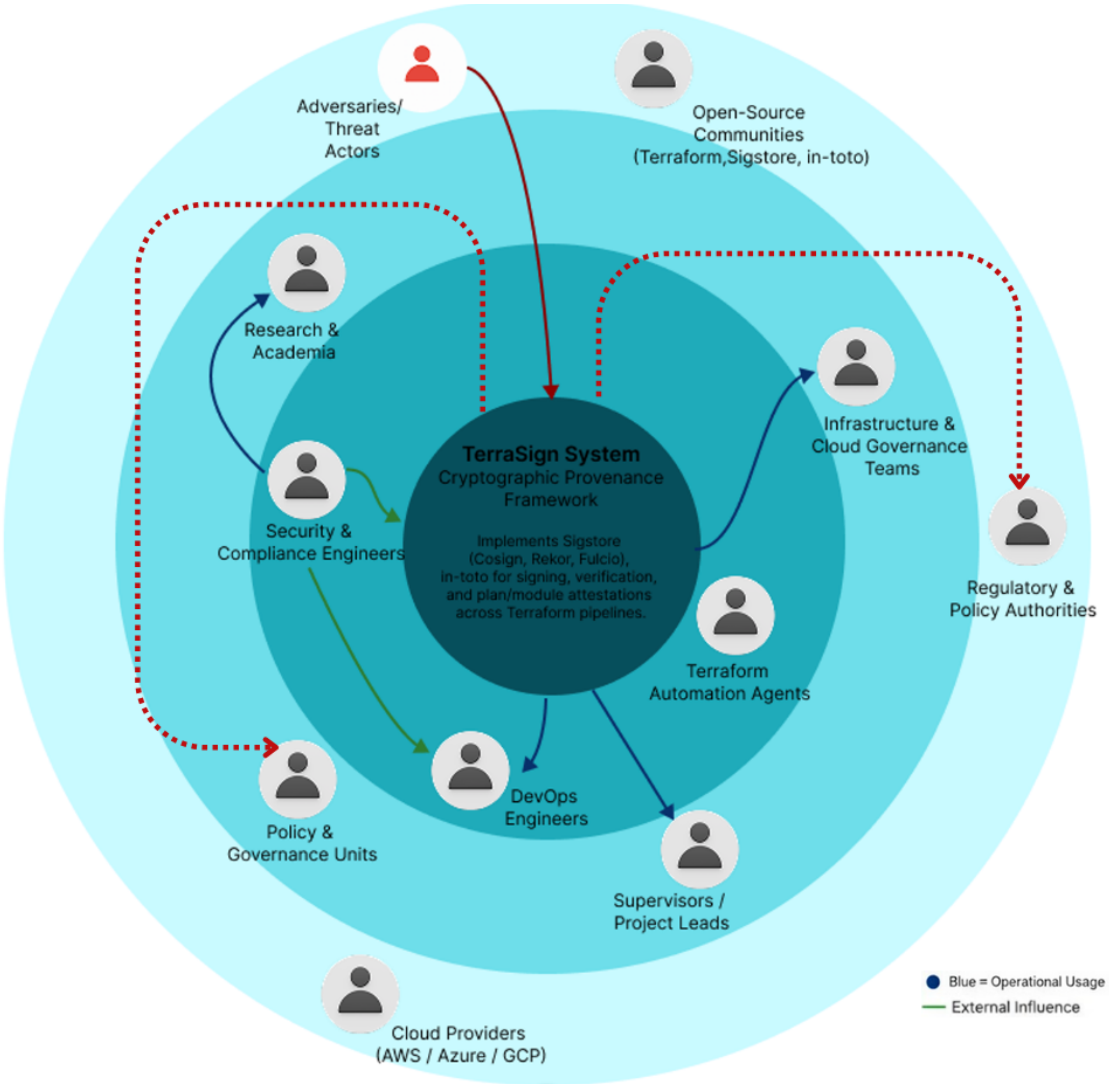


Figure 6: Stakeholder Onion Model (Self-composed)

4.3.2 Stakeholder Viewpoints

Stakeholder	Role	Viewpoint
DevOps Engineers / CI-CD Pipelines	Primary system users	Seek a seamless, automated integration of module signing and verification into existing Terraform workflows. Expect minimal disruption, faster deployments, and clear feedback during provenance validation.
Security & Compliance Engineers	System integrity enforcers	Require reliable, verifiable provenance records to ensure only trusted and signed Terraform modules are deployed. Expect granular logs, alerts, and policy-driven access control to maintain auditability.
Terraform Automation Agents	Operational executors	Act as automated agents performing continuous verification and attestation during pipeline runs. Require stable APIs, minimal latency, and uninterrupted signing workflows.
Supervisors / Project Leads	Governance and oversight	Ensure that deployment processes align with organizational security objectives. Expect system reports and dashboards that summarize verification status and compliance metrics.
Infrastructure & Cloud Governance Teams	Policy enforcers and custodians	Focus on applying cryptographic provenance policies across multi-cloud environments. Require integration with AWS/Azure/GCP for provenance validation and compliance logging.
Policy & Governance Units	Regulatory alignment and compliance	Define trusted roots of signing and verification authority. Expect TerraSign to align with frameworks such as **SLSA Level 3+**, **NIST SP 800-204D**, and internal DevSecOps trust models.

Research & Academia	Domain experts and evaluators	Investigate TerraSign's impact on software supply-chain security. Seek access to anonymized data for academic evaluation, comparison, and research on secure IaC provenance.
Cloud Providers (AWS / Azure / GCP)	Infrastructure hosts	Provide environments where signed Terraform modules are deployed. Expect interoperability with provenance validation mechanisms and attestation logs.
Open-Source Communities (Terraform, Sigstore, in-toto)	External contributors	Maintain and evolve core provenance, attestation, and signing tools that TerraSign relies upon. Expect community collaboration and contribution transparency.
Regulatory & Policy Authorities	External compliance bodies	Ensure that organizational IaC deployments meet evolving regulatory and cybersecurity standards. Expect evidence of traceability, non-repudiation, and compliance documentation.
Adversaries / Threat Actors	Negative stakeholders	Attempt to inject malicious or unverified Terraform modules or tamper with provenance records. Their perspective highlights the importance of cryptographic signing, transparency logs, and active verification defenses.

Table 8: Stakeholder Viewpoints

4.4 Selection of Requirement Elicitation Methodologies

To capture comprehensive and realistic requirements, we employed multiple requirement elicitation methodologies,

4.4.1 Literature Review

This report reviewed a variety of academic literature, industry reports, and practitioner reports related to Infrastructure as Code (IaC) security, supply-chain vulnerabilities, and cryptographic provenance. The review summarized the knowledge about the known threats and countermeasures, including the OWASP CI/CD risk catalogue and previous empirical research on Terraform security. The literature offered a methodical synthesis of theoretical preconditions as venerated by domain experts, arranging the resultant conditions in line with modern security frameworks, such as the Security Levels for Supply-chain Assurance (SLSA) model, and with tooling ecosystems such as Sigstore and in-toto, which offers cryptographic attestation services.

4.4.2 Surveys

A purposive group of Terraforms practitioners and the managers of continuous integration/continuous delivery (CI/CD) pipelines were surveyed with an online questionnaire. The instrument was used to evaluate current practices, e.g., checking the safety of modules, knowledge of signing procedures, willingness to implement extra protective measures, and perceived facilitators and obstacles by combining discrete-choice questions with open-ended questions. Quantitative answers were measures of current trends, including the percentage of teams that are performing module integrity checks and the confidence level in modules that have cryptographic signatures. Qualitative feedback explained contextual issues in the adoption of potential security measures.

4.4.3 Brainstorming and Observation.

On the inside, brainstorming sessions were organized systematically to sketch the Terraform lifecycle, starting with the initialization and planning stage up to actual execution, and finding points where provenance metadata could be produced or verified. Simultaneously, an observational study of a test-bed pipeline was used to capture the nature of external module retrieval and vulnerabilities, such as the possibility of a module to be changed between plan and apply phases without explicit locking, which was supported by Proulx (2023). Artificial adversarial events, including post-planning modules replacement, were simulated in order to reveal functional requirements necessary to identify tampering. The session also discussed the possibility of incorporating such tooling as Cosign and in-toto and suggested few changes in the pipeline and considering the burden on developers.

The combination of the various methodologies led to a set of requirements. Literature provides a high-level framework of justification of such features as the mandatory cryptographic validation at all the steps of the pipeline, whereas surveys and interviews provide real-life constraints and expectations of the stakeholders. These insights are then converted into practical technical specifications in brainstorming and pipeline observation, and edge cases that are not visible otherwise. This combined methodology will increase the confidence that the ultimate Software Requirements Specification will both meet the theoretical security requirements and meet the practical needs of the users.

4.5 Discussion of Findings

For each methodology used, key findings are documented below.

4.5.1 Literature Review Findings

Findings	Citations
Terraform modules lack built in cryptographic verification. Modules in public registries can be tampered if referenced by mutable version tags.	(Proulx, 2023)
OWASP classifies supply chain attacks as top CI/CD threats and recommends cryptographic validation across the pipeline.	(OWASP, 2023)
Industry platforms like npm and Microsoft are adopting provenance and signing tools like Sigstore to secure build artifacts.	(Microsoft, 2023; Node.js Foundation, 2023)
Terraform Registry updates now tie module versions to Git SHAs, but this does not apply to private modules or enforce universal provenance.	(HashiCorp, 2024)
Provenance frameworks like SLSA have shown to reduce attack risks by enforcing source integrity and recording build steps.	(Tamanna et al., 2024)
Real-world SLSA adoption reveals barriers like lack of tooling and complexity. Simplicity and automation are needed for adoption.	(Tamanna et al., 2024)

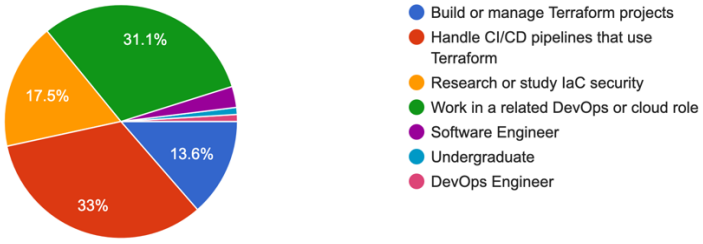
Terraform projects often skip basic security best practices. Static analysis tools do not validate code origin, leaving a trust gap.	(Verdet et al., 2023)

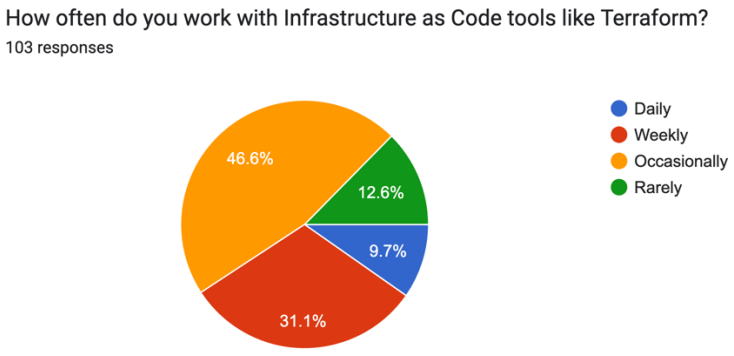
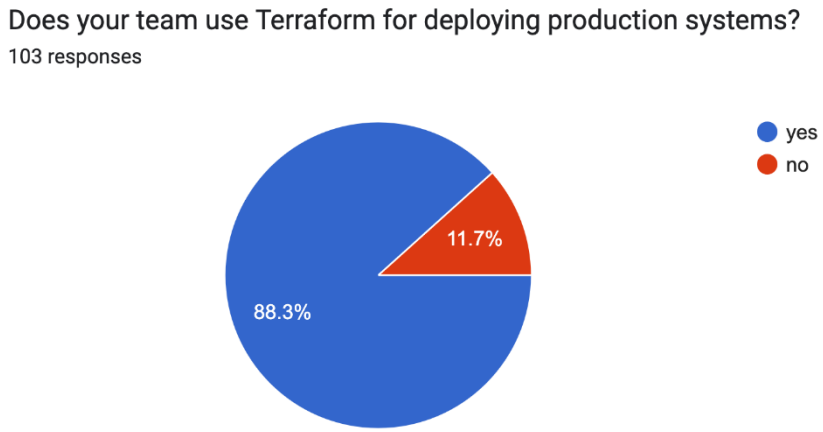
Table 9: Literature Review Findings

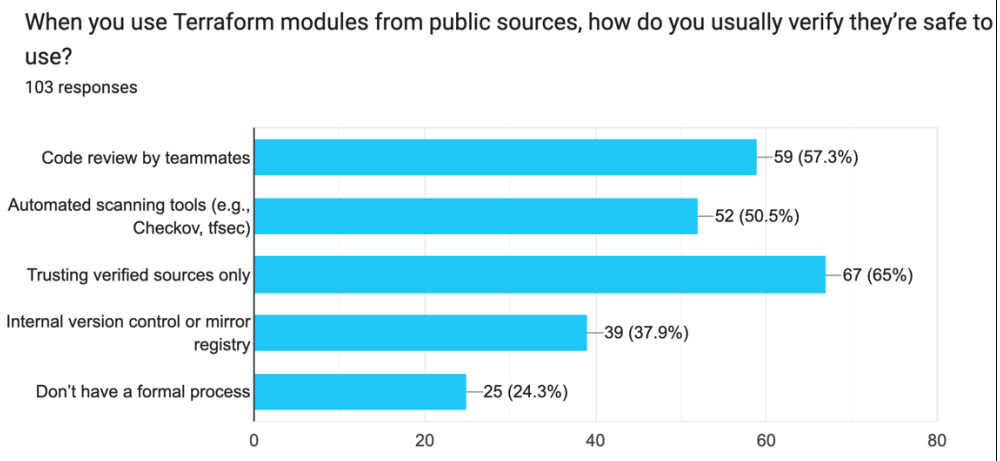
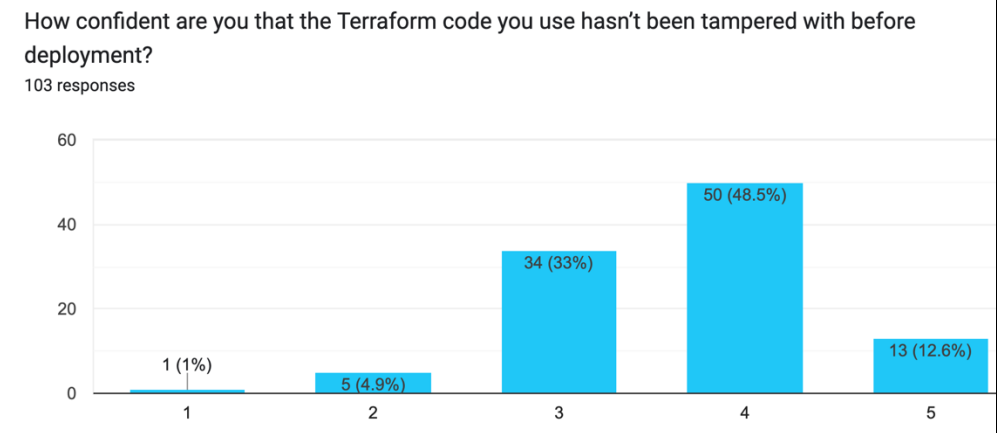
4.5.2 Survey Findings

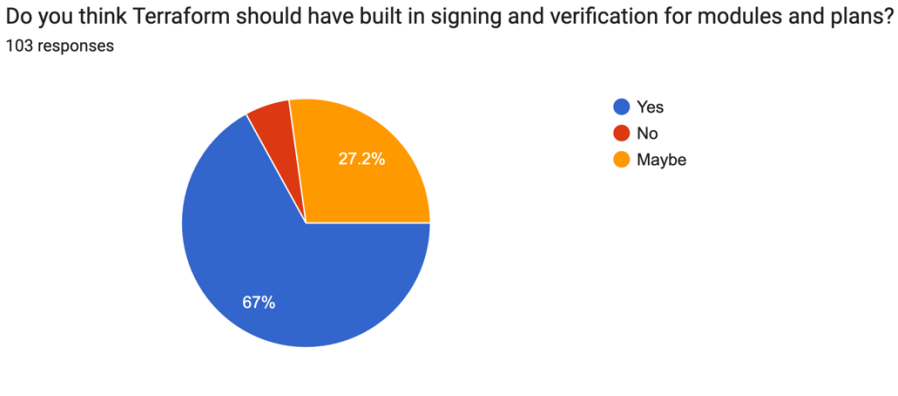
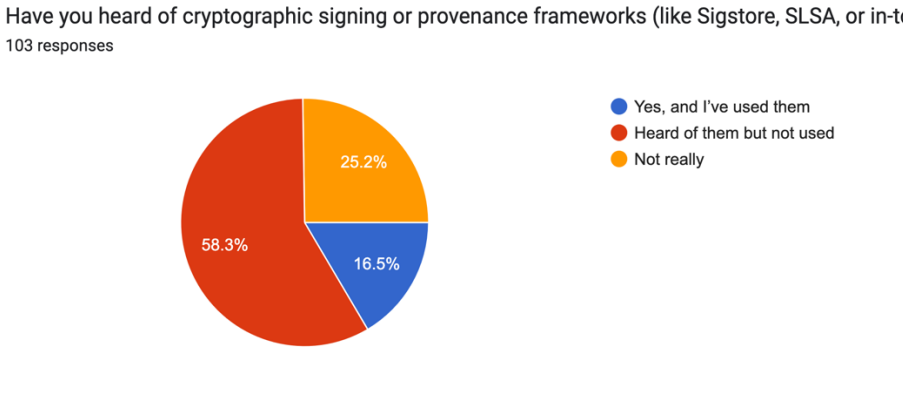
Online survey was structured to capture the system expectations from x, y, and z stakeholders. Refer to questionnaire in Appendix A. The online questionnaire was sent for <sent number> where a total of <received responses> were received.

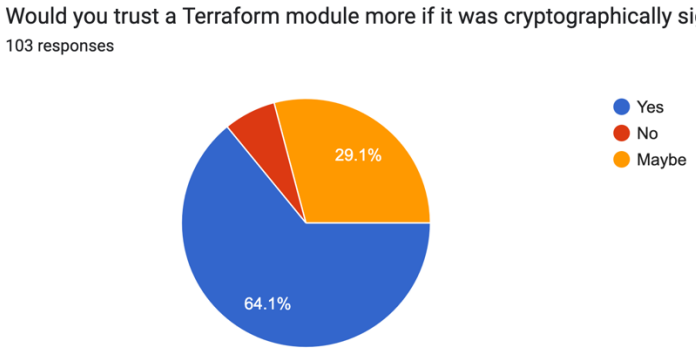
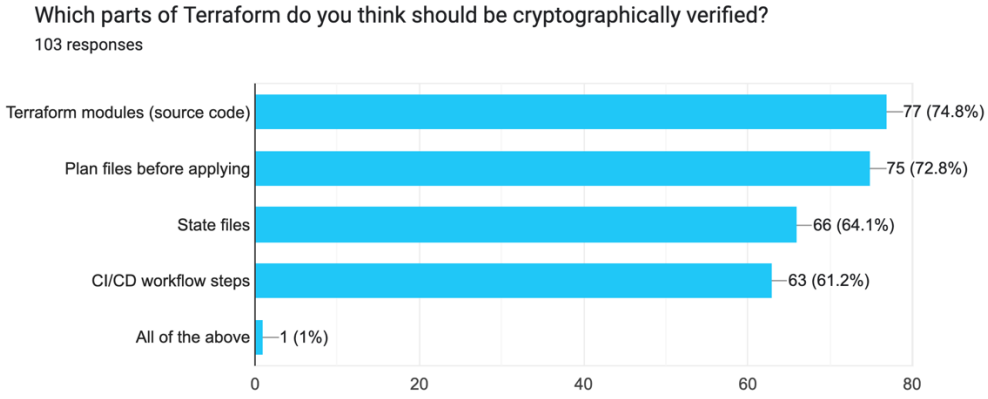
4.5.3 Response Analysis

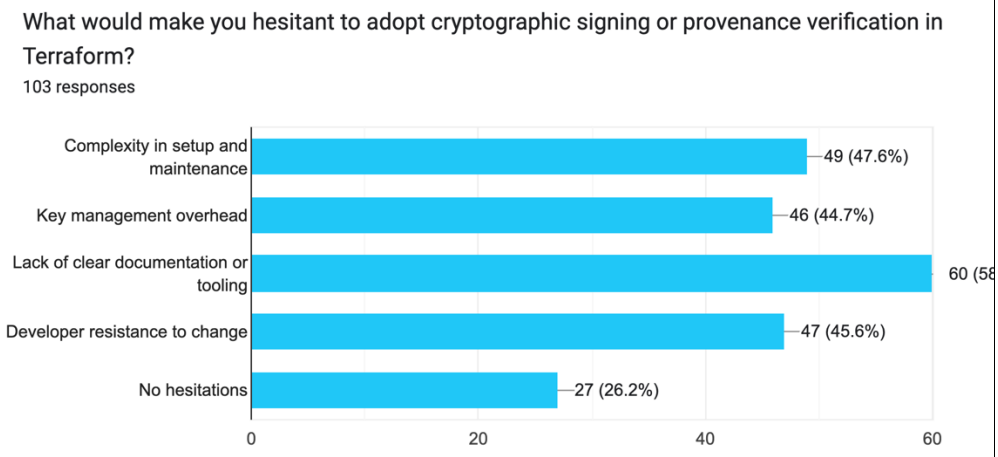
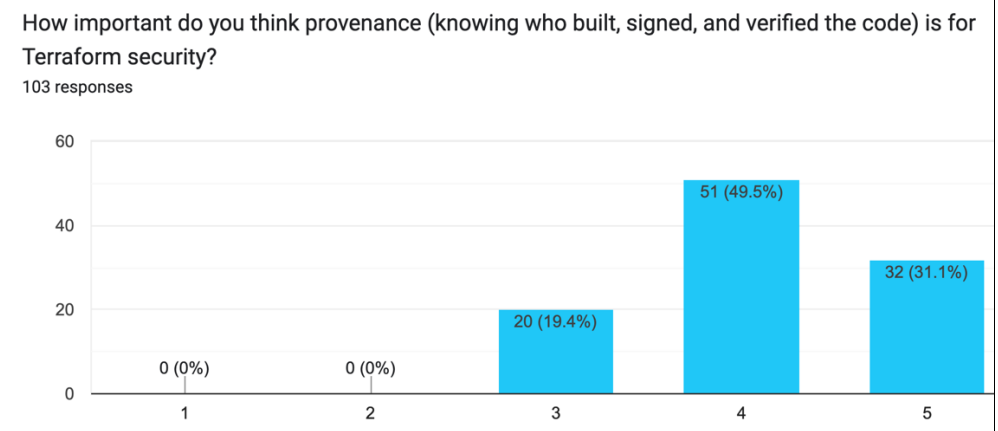
Section 1	
Question 1.1	How are you currently involved with Terraform or Infrastructure as Code (IaC)?
Aim of the Question	To identify how frequently users interact with Terraform.
Findings	 Most participants use Terraform weekly. Daily and occasional use are also common.
Question 1.1	How often do you work with Infrastructure as Code tools like Terraform?
Aim of the Question	To understand if Terraform is used in production environments.

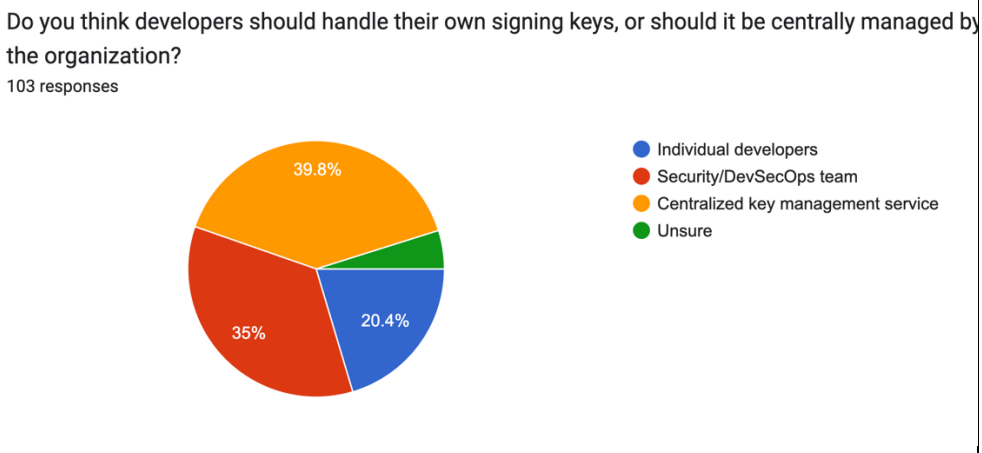
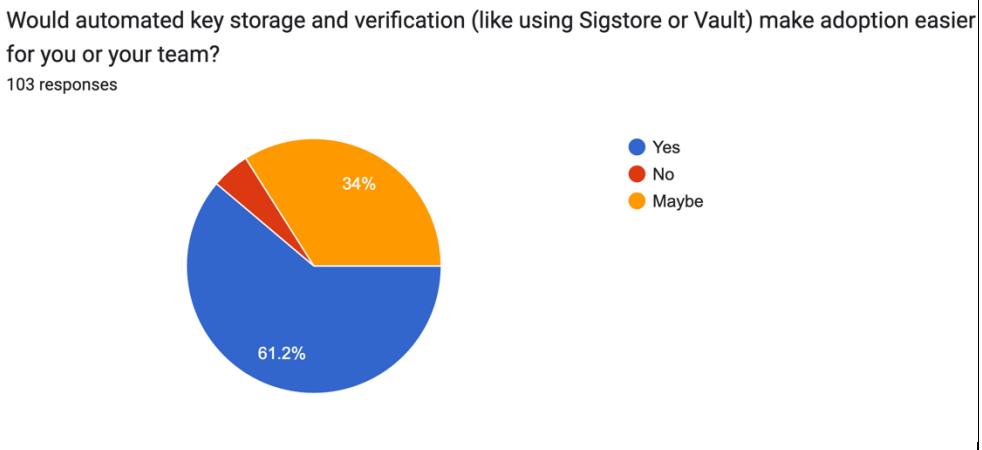
Findings	 Few use Terraform in production. Most use it in non-production environments.
Question 1.1	Does your team use Terraform for deploying production systems?
Aim of the Question	To check whether Terraform is used for deploying production.
Findings	
Question 1.1	When you use Terraform modules from public sources, how do you usually verify they're safe to use?
Aim of the Question	To identify common methods used for module trust verification.

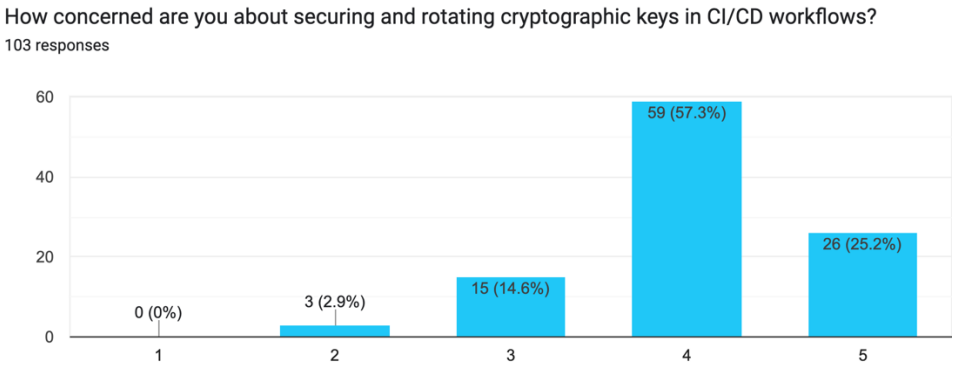
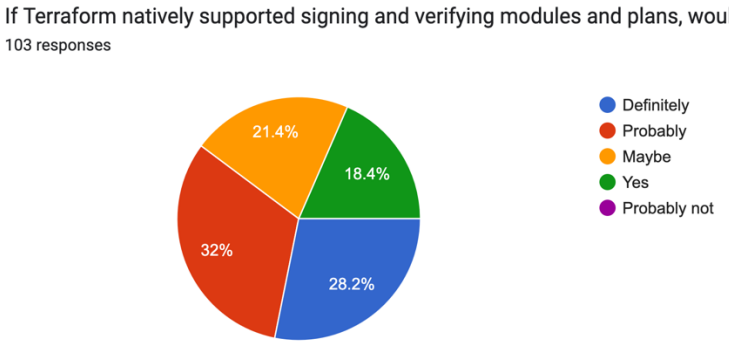
Findings	 Users rely on trusted sources, scanning tools, and code review for verification.
Question 1.1	How confident are you that the Terraform code you use hasn't been tampered with before deployment?
Aim of the Question	To assess confidence in Terraform code integrity.
Findings	 Most feel confident in their code's integrity despite minimal formal checks.
Question 1.1	Do you think Terraform should have built in signing and verification for modules and plans?
Aim of the Question	To assess interest in native signing features in Terraform.

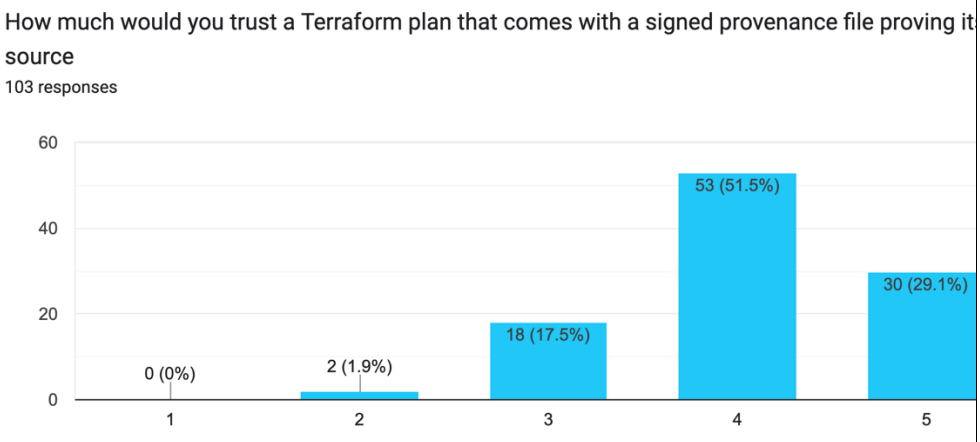
Findings	 Majority want Terraform to support built-in signing and verification.
Question 1.1	Have you heard of cryptographic signing or provenance frameworks (like Sigstore, SLSA, or in-toto)?
Aim of the Question	To measure awareness of cryptographic provenance tools.
Findings	 Most have not heard of or used signing tools like Sigstore or SLSA.
Question 1.1	Would you trust a Terraform module more if it was cryptographically signed by the author?
Aim of the Question	To assess the effect of cryptographic signing on user trust.

Findings	 Trust in modules increases when cryptographic signatures are present.
Question 1.1	Which parts of Terraform do you think should be cryptographically verified?
Aim of the Question	To determine which parts of the Terraform workflow need verification.
Findings	 Modules, plans, state files, and CI/CD steps are key verification targets.
Question 1.1	What would make you hesitant to adopt cryptographic signing or provenance verification in Terraform?
Aim of the Question	To identify challenges that discourage adoption of signing practices.

Findings	 Tooling gaps, complexity, and key management are top barriers to adoption.
Question 1.1	How important do you think provenance (knowing who built, signed, and verified the code) is for Terraform security?
Aim of the Question	To understand perceived value of provenance in Terraform security.
Findings	 Provenance is considered important by most DevOps practitioners.
Question 1.1	Do you think developers should handle their own signing keys, or should it be centrally managed by the organization?
Aim of the Question	To explore preferences for key management responsibility.

Findings	 Centralized or team-based key management is preferred over individual control.
Question 1.1	Would automated key storage and verification (like using Sigstore or Vault) make adoption easier for you or your team?
Aim of the Question	To measure openness to adoption of built-in signing features.
Findings	 Most users would adopt native signing if it became available.
Question 1.1	How concerned are you about securing and rotating cryptographic keys in CI/CD workflows?
Aim of the Question	To assess how users perceive the risks and management burden of cryptographic keys.

Findings	 Most participants see key management as a major operational concern in implementing signing.
Question 1.1	If Terraform natively supported signing and verifying modules and plans, would your team adopt it?
Aim of the Question	To measure actual adoption interest based on availability of native support.
Findings	 Most participants show positive intent to adopt if Terraform includes native signing features.
Question 1.1	How much would you trust a Terraform plan that comes with a signed provenance file proving its source
Aim of the Question	To understand how signing affects trust in deployment plans.

Findings	 Participants express high trust in signed plans compared to unsigned ones.
Question 1.1	In your opinion, what's the biggest challenge with implementing signing and verification in real world Terraform projects?
Aim of the Question	To collect direct feedback on the most difficult part of deploying provenance checks
Findings	I would think existing teams would be bit hesitant to adapt considering the licencing and migration efforts. I think you should consider opentofu as well on this. As well as there is azure verified modules as of now but they are yet not being adapter widely 1 response In my opinion operational complexity of managing and integrating the required security processes 1 response In my opinion, the biggest challenge with implementing signing and verification in real-world Terraform projects is the complexity of managing the custom provider and module signing lifecycle within an automated CI/CD workflow 1 response

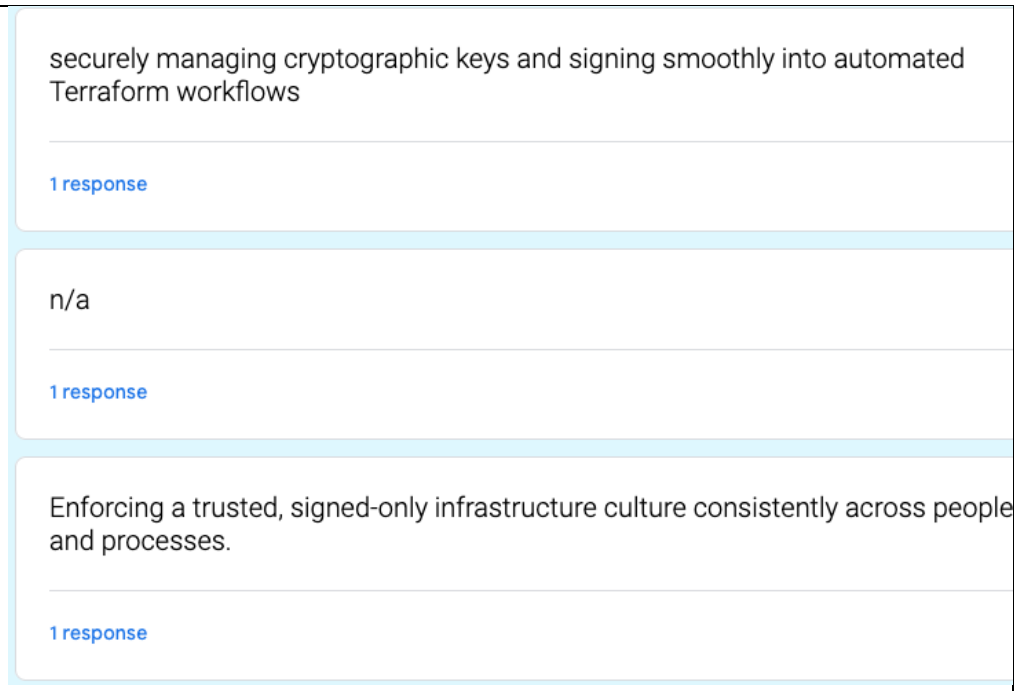
	 Most responses mention enforcement, automation, and pipeline integration as the hardest parts.

Table 10: Response Analysis of the Survey

4.5.4 Brainstorming/Observation

Criteria	Findings
Terraform Pipeline Touchpoints	Identified key stages in Terraform where signing and verification must occur, specifically at module fetch and pre-apply. These are critical to prevent post plan tampering.
Feasibility of Using Sigstore and in-toto	Confirmed Sigstore and in-toto tools are suitable for signing modules and recording provenance. Proposed integration with Fulcio and Rekor, with optional support for in-toto attestations.
Module Tampering Simulation	Simulated tampering showed Terraform alone does not detect changes. Prototype signing approach successfully flagged the mismatch. Requirement added for mandatory hash verification.

Identification of External Interfaces	Mapped out system interactions with Terraform CLI, CI/CD runners, registries, Fulcio, Rekor, and internal tools. Requirement added for compatibility with multiple module sources and offline verification.
Brainstorming Security Policy	Designed flexible trust model. Requirement added to support configurable security policies for accepted signers and trust roots, improving enterprise usability.

Table 11: Brainstorming Findings

4.6 Summary of Findings

Findings	**Surveys**	**Literature Review**	**Brainstorming/Observation**
Users expect cryptographic verification for Terraform modules and plans.	✓	✓	✓
Provenance and artifact trust are seen as critical to Terraform security.	✓	✓	✓
Sigstore and in-toto are viable tools for implementing provenance.		✓	✓
Module tampering is a real risk in Terraform's current model.	✓	✓	✓
Users prefer automated and simple solutions over complex setups.	✓		✓
Key management and verification policy must be flexible and organization-controlled.	✓	✓	✓
Support for offline or private registry signing is needed.		✓	✓
Verification should occur at key Terraform steps: module fetch and apply.		✓	✓
Awareness of signing tools remains low despite interest in secure pipelines.	✓		

Table 12: Summary of Findings

4.7 Context Diagram

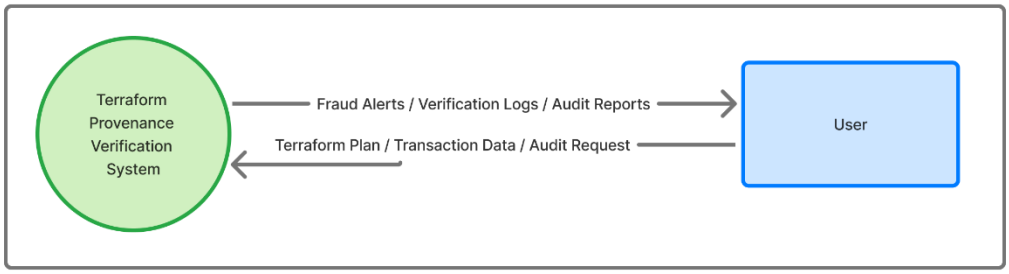


Figure 7: Context Diagram (Self-composed)

4.8 Use Case Diagram

The **Use Case Diagram** represents the key interactions between the system and its users. The actors in the **Use Case Diagram** should match those in the **Context Diagram** to maintain

Main Actors:

- **Fraud Analysts**: Receive fraud alerts.
- **End Users**: Make transactions.
- **Regulatory Bodies**: Review compliance reports.

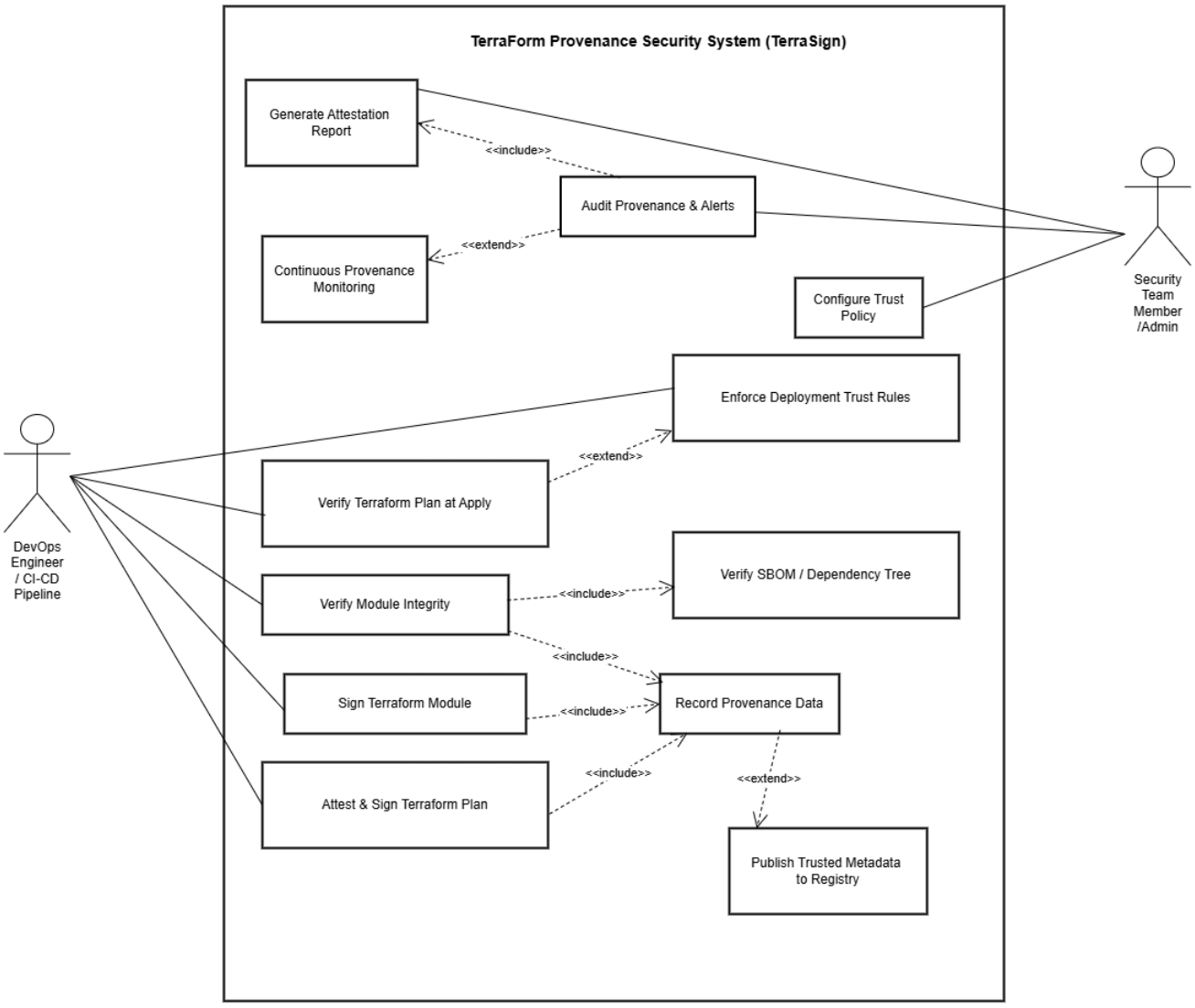


Figure 8: Use case Diagram (Self-composed)

4.9 Use Case Descriptions

Use case	Sign Terraform Module
Id	UC:01
Description	The DevOps Engineer initiates the signing process during Terraform plan or apply operations. The system intercepts downloaded modules, computes their cryptographic hash, and applies a digital signature. The generated

	signature is then stored in the internal provenance log for traceability and audit purposes.
Actor	DevOps Engineer / CI-CD Pipeline
Supporting actor	None
Stakeholders	DevOps Team, Infrastructure Security Team
Preconditions	Terraform execution environment and modules are accessible. System trust policy is preconfigured with acceptable signing certificates.
Main flow	• DevOps Engineer triggers a Terraform plan or apply command. • System intercepts the module retrieval process. • A unique hash is generated for each module. • The system signs the module using the configured private key. • The signed module and associated metadata are recorded in the provenance log.
Alternative flows	If module hash computation fails, the system retries with fallback hashing algorithm. If signature creation fails, the module is skipped and logged as unsigned.
Exceptional flows	If system cannot access the signing key or log storage, Terraform execution halts with an integrity warning.
Postconditions	Signed module records are securely stored and available for future verification.

Table 13: Use Case Descriptions UC:01

4.10 Requirements

4.10.1 Functional Requirements

ID	Requirement	Priority (MoSCoW)
FR01	**Real-time Fraud Detection:** The system **must** detect fraudulent transactions in real-time, as transactions are being processed. *Explanation:* This means as soon as a transaction request comes in, the system's fraud detection engine will analyze it on the fly and decide whether to approve or flag it within seconds.	Must Have
FR02	**Alert Generation:** The system **must** provide immediate fraud alerts to fraud analysts when a suspicious transaction is detected. *Explanation:* For any transaction flagged as potentially fraudulent (by FR01), an alert/notification is created containing details of the transaction (and risk reasons) and is sent to the queue or dashboard for analysts to review.	Must Have
FR03	**Analyst Override:** The system **should** allow fraud analysts to override a fraud decision (false positive) in case a legitimate transaction is wrongly flagged. *Explanation:* Via the interface, an analyst can mark an alert as "Not Fraud" which will reverse any automatic hold on that transaction and log the override. This trains the system or at least records feedback for future improvements.	Should Have
FR04	**External Data Integration:** The system **could** integrate with external databases or APIs for additional fraud checks. *Explanation:* For example, it could query a national blacklist database or use credit score services during its analysis. This can enhance detection accuracy by providing more information, but is considered an enhancement that could be added when core features are stable.	Could Have

Table 14: Functional Requirements

4.10.2 Non-Functional Requirements

ID	Non-Functional Requirement	Description	Priority (MoSCoW)
NFR01	Scalability	The system **must** handle a high throughput of transactions, specifically at least **1,000 transactions per second** during peak loads. *Rationale:* The bank anticipates heavy volumes, and the fraud checks should scale (horizontally, if needed) to analyze each transaction without performance degradation.	Must Have
NFR02	Security	The system **must** encrypt all sensitive transaction data both in transit and at rest. *Rationale:* This includes using protocols like HTTPS/TLS for data in motion and strong encryption (AES-256 or better) for stored data (databases, logs). This requirement ensures compliance with security regulations and protects customer information from breaches.	Must Have
NFR03	Performance	The system **should** maintain an average response time of **less than 1 second** for transaction analysis. *Rationale:* This means from the moment a transaction is submitted to the fraud engine to the moment a decision (fraud/not fraud) is returned, under 1 second should elapse on average. This ensures that legitimate transactions are not unduly delayed and real-time detection (FR01) truly feels instantaneous to users.	Should Have

NFR04	Usability	The system **could** provide an intuitive dashboard interface for real-time monitoring of transactions and alerts by analysts. *Rationale:* Although not critical for core functionality, a well-designed UI (graphs of transaction flow, alert filters, etc.) would greatly enhance analyst efficiency and satisfaction. This could include features like color-coded alerts by severity, search and filter capabilities, and user-friendly workflows for investigating and resolving alerts.	Could Have

Table 15: Non-Functional Requirements

4.11 Chapter Summary

In this chapter, the stakeholders who are DevOps engineers and the security team are identified and analyzed with respect to perspectives about the proposed system. The requirements are gathered with the use of various requirement gathering techniques, such as extensive research gathered from the study carried out in the past about various research works already carried out to solve similar problems and industry practices to identify what is already required without affecting user expectations, with extensive analysis carried out within the team about the Terraform workflows to identify areas to integrate. The analysis gathered essential requirements, such as the absence of end-to-end provenance solutions in the current settings, user requirements to automate with minimal disruption, and performance requirements to allow functioning in production settings. The chapter then defined the system context by mapping all external interfaces and interactions. The chapter then continued with use cases that illustrated the functioning of the entire system with respect to various operational settings. The chapter concluded with comprehensive requirements that are prioritized with the MoSCoW analysis to define what needs to be achieved by the entire system to ensure the Terraform supply chain with cryptographic provenance while still ensuring productivity among DevOps engineers.

Time Schedule

Include here the Updated Gantt chart where you show the actual progress upto PPRS submission.

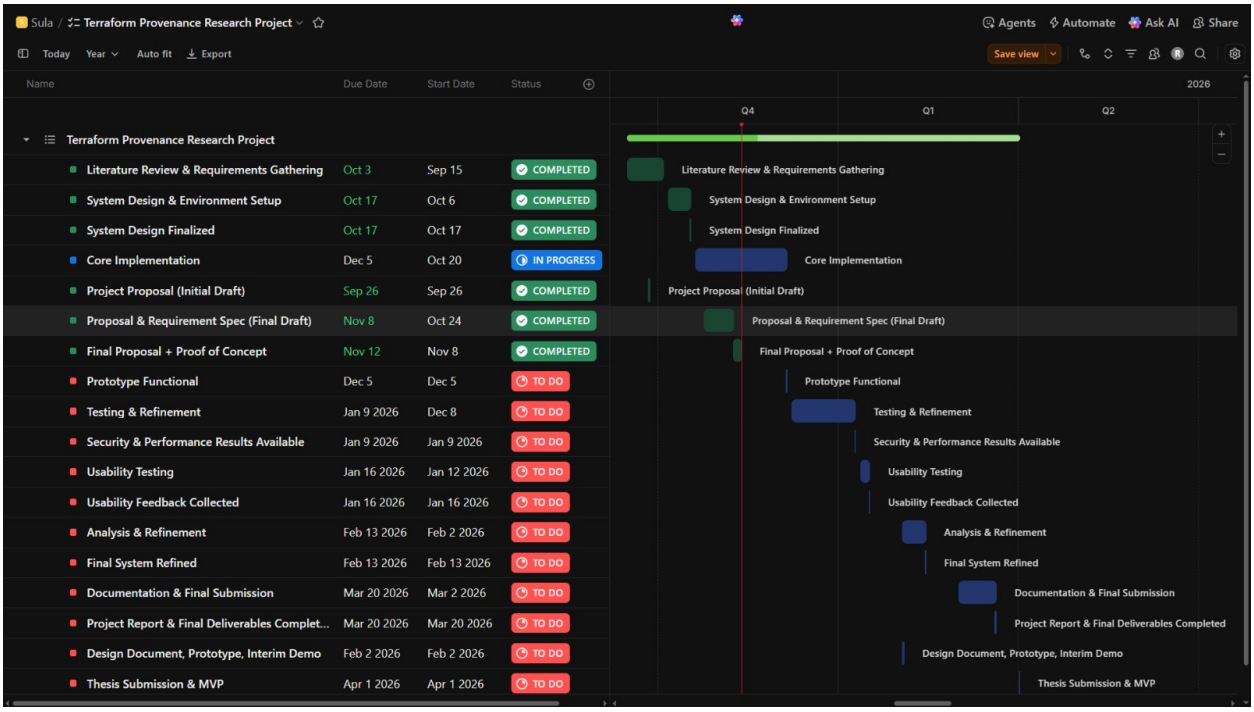


Figure 9: Gantt Chart(Self-composing)

References

Douglas, N. (2023) Terraform Security Best Practices. Sysdig Blog, 21 March 2023. Available at: https://www.sysdig.com/blog/terraform-security-best-practices (Accessed Jun. 2025).

Eisenkot, G. (2022) Prisma Cloud Supply Chain Security Reduces Code Complexity and Risk. Palo Alto Networks Blog Available at: https://www.paloaltonetworks.com/blog/2022/03/cloud-software-supply-chain-security/ (Accessed Jun. 2025).

HashiCorp (2024) HCSEC-2024-04: Terraform Registry Module Supply Chain Security Improvements. HashiCorp Discuss forum. Available at: https://discuss.hashicorp.com/t/hcsec-2024-04-terraform-registry-module-supply-chain-security-improvements/62815 (Accessed Jul. 2025).

Igboanugo, D.U. (2025) CI/CD in the Age of Supply Chain Attacks: How to Secure Every Commit. DZone (Sep. 4, 2025). Available at: https://dzone.com/articles/ci-cd-pipeline-security-supply-chain (Accessed Jul. 2025).

Lepiller, J., Piskac, R., Schäf, M. and Santolucito, M. (2021) Analyzing Infrastructure as Code to Prevent Intra-Update Sniping Vulnerabilities. In Lecture Notes in Computer Science, 12652, pp. 105–123. Springer. Available at: https://link.springer.com/chapter/10.1007/978-3-030-72013-1_6 (Accessed Aug. 2025).

Microsoft (2022) What Is DevSecOps? Definition and Best Practices. Microsoft Security Blog. Available at: Microsoft DevSecOps. Available at: https://www.microsoft.com/en-us/security/business/security-101/what-is-devsecops (Accessed Aug. 2025).

OWASP (2023) OWASP Top 10: CI/CD Security Risks. OWASP Foundation. Available at: https://owasp.org/www-project-top-10-ci-cd-security-risks/ (Accessed Oct. 2025).

Proulx, F. (2023) Erosion of Trust: Unmasking Supply Chain Vulnerabilities in the Terraform Registry. BoostSecurity Blog. Available at: https://boostsecurity.io/blog/erosion-of-trust-unmasking-supply-chain-vulnerabilities-in-the-terraform-registry (Accessed Jul. 2025).

Reed, M. (2022) HashiCorp Terraform Code Signing. Green Reed Technology Blog. Available at: https://www.greenreedtech.com/hashicorp-terraform-code-signing/ (Accessed Oct. 2025).

Schneeweisz, J. (2022) Terraform in the Software Supply Chain: Modules & Providers. GitLab Blog. Available at: https://about.gitlab.com/blog/terraform-as-part-of-software-supply-chain-part1-modules-and-providers/ (Accessed Jun. 2025).

Sigstore (2023) Sigstore Support in npm Launches for Public Beta. Sigstore Project Blog (Apr. 19, 2023). Available at: https://blog.sigstore.dev/npm-public-beta/ (Accessed Aug. 2025).

Sirish, A. and Kennedy, C. (2023) Unleashing in-toto: The API of DevSecOps. CNCF Blog (Aug. 17, 2023). Available at: https://www.cncf.io/blog/2023/08/17/unleashing-in-toto-the-api-of-devsecops/ (Accessed Oct. 2025).

Tamanna, M., et al. (2024) Unraveling Challenges with Supply-Chain Levels for Software Artifacts (SLSA) for Securing the Software Supply Chain. arXiv:2409.05014 [cs.CR]. Available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4979511 (Accessed Jul. 2025).

Tran, A.-D., Sion, L., Yskout, K. and Joosen, W. (2025) TerrARA: Automated Security Threat Modeling for Infrastructure as Code. In Proceedings of the 2025 ACM Conference on Data and Application Security and Privacy (CODASPY '25), pp. 269–280. Available at: https://www.researchgate.net/publication/392427008_TerrARA_Automated_Security_Threat_Modeling_for_Infrastructure_as_Code (Accessed Aug. 2025).

Verdet, A., Hamdaqa, M., da Silva, L. and Khomh, F. (2023) Exploring Security Practices in Infrastructure as Code: An Empirical Study. arXiv:2308.03952 [cs.SE]. Available at: https://arxiv.org/abs/2308.03952 (Accessed Jun. 2025)

Appendix

Use case	Verify Module Integrity
Id	UC:02
Description	Before deployment, the system verifies the authenticity and integrity of Terraform modules. The verification process compares module digests and signature chains against trusted provenance records to ensure no tampering occurred.
Actor	DevOps Engineer / CI-CD Pipeline
Supporting actor	None
Stakeholders	DevOps Team, Security Compliance Division
Preconditions	• Signed provenance records exist for all referenced modules. • Trust policy and signature verification components are operational.
Main flow	• DevOps Engineer triggers Terraform apply. • System retrieves stored signature and digest from provenance log. • Current module hash is computed and matched against recorded hash. • Verification results determine whether module is accepted or rejected.
Alternative flows	If provenance record is unavailable, system fetches from backup log storage.
Exceptional flows	If hash mismatch occurs, the deployment process stops, and the engineer is notified of potential tampering.
Postconditions	Verified modules proceed to deployment; unverified modules are blocked

Table 16: Use Case Descriptions UC:02

Use case	Attest and Sign Terraform Plan
Id	UC:03
Description	After generating a Terraform plan, the system creates an attestation to confirm its authenticity and approval. This ensures the plan being applied later has not been modified.
Actor	DevOps Engineer
Supporting actor	None
Stakeholders	DevOps Team, Release Management
Preconditions	• Terraform plan must be successfully generated. • Signing and attestation components are active.
Main flow	• Engineer generates a Terraform plan. • System captures plan summary and metadata. • The plan is signed and stored in the provenance database. • System links attestation to the corresponding configuration commit.
Alternative flows	If plan signing fails, system prompts reattempt or skips attestation with warning.
Exceptional flows	Missing signing certificate or corrupt plan file halts the process.
Postconditions	Terraform plan attestation stored for future verification before apply stage.

Table 17: Use Case Descriptions UC:03

Use case	Verify Terraform Plan at Apply
Id	UC:04
Description	Before applying changes, the system ensures the current Terraform plan matches the previously signed and attested plan.
Actor	DevOps Engineer / CI-CD Pipeline
Supporting actor	None
Stakeholders	DevOps Team, Infrastructure Security
Preconditions	Attestation record for the plan exists in provenance logs.
Main flow	• Engineer initiates Terraform apply. • System retrieves attestation record. • Current plan hash is compared with attested plan hash. • If matched, apply continues; else system enforces block.
Alternative flows	System retries from backup provenance copy if local attestation not found.
Exceptional flows	Hash mismatch or missing record causes immediate stop and security alert.
Postconditions	Ensures only approved plans are deployed to infrastructure.

Table 18: Use Case Descriptions UC:04

Use case	Record Provenance Data
Id	UC:05

Description	The system continuously logs module, plan, and signature information to create a verifiable provenance trail for all Terraform operations.
Actor	System (Automated)
Supporting actor	DevOps Engineer
Stakeholders	Security Compliance, Audit Teams
Preconditions	Provenance storage backend (database or log file) must be available.
Main flow	• Each signing and verification action triggers a provenance entry. • System records identity, timestamp, module digest, and outcome. • Logs are sealed to prevent retroactive tampering.
Alternative flows	In offline mode, entries are cached locally until reconnection.
Exceptional flows	Failure to write logs triggers warning and marks current run as non-auditable.
Postconditions	Immutable provenance records available for future audits and compliance.

Table 19: Use Case Descriptions UC:05

Use case	Audit Provenance and Alerts
Id	UC:06
Description	Security Team members review historical provenance data, analyze verification outcomes, and investigate failed module checks or untrusted plans.
Actor	Security Team Member

Supporting actor	None
Stakeholders	Security Operations Center, Compliance Department
Preconditions	Provenance records are up-to-date and accessible.
Main flow	• Security analyst requests provenance audit. • System aggregates relevant signing and verification events. • Analyst reviews summary reports and alert logs. • Non-compliance cases are flagged for remediation.
Alternative flows	Analyst applies filters (by date, environment, or signer) for detailed analysis.
Exceptional flows	Missing or corrupted provenance entries generate integrity warnings.
Postconditions	Verified audit trail supports compliance and internal reviews.

Table 20: Use Case Descriptions UC:06

Use case	Configure Trust Policy
Id	UC:07
Description	Security administrators define which signers, certificates, and domains are trusted for Terraform module and plan verification.
Actor	Security Administrator
Supporting actor	None
Stakeholders	Cloud Governance, Infrastructure Security

Preconditions	Administrative access and authentication verified.
Main flow	• Admin accesses trust policy configuration interface. • Adds or removes trusted signers and certificate authorities. • Updates rules defining required verification levels (e.g., strict, permissive). • Saves and propagates configuration across the system.
Alternative flows	If new policy conflicts with existing one, system prompts review before saving.
Exceptional flows	Invalid certificate format or unreachable CA endpoint halts policy update.
Postconditions	Updated trust policy governs future signing and verification behavior.

Table 21: Use Case Descriptions UC:07

Survey link

https://forms.gle/34RGMPfT2ywnLMfy7