International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I2P126
Eureka Vision Publication | Volume 7, Issue 2, 191-202, 2026

Original Article

Infrastructure as Code Security: Static Analysis and Policy Enforcement for Terraform and Ansible Deployments

Bharat Singh Chaudhary
Independent Researcher Cloud Security & DevSecOps Architect, Quorum Information Technology, Calgary, Alberta, Canada.

Received On: 14/03/2026 ***Revised On: 13/04/2026*** ***Accepted On: 20/04/2026*** ***Published On: 27/04/2026***

Abstract - Infrastructure as Code (IaC) has fundamentally changed how organizations provision and manage cloud resources. Terraform and Ansible are the two most widely adopted IaC tools, with Terraform dominating declarative infrastructure provisioning and Ansible leading configuration management and application deployment. The shift from manual infrastructure management to code-defined infrastructure introduces a new class of security risks: misconfigurations encoded in version control that propagate at scale across every deployment. A single misconfigured Terraform module can create hundreds of publicly accessible storage buckets, unencrypted databases, or overly permissive network rules across multiple cloud accounts within minutes of a terraform apply. This paper examines the security risks inherent in IaC workflows and proposes a multi-layered security framework combining pre-commit static analysis, CI pipeline scanning with plan-level evaluation, policy-as-code enforcement using Open Policy Agent and Sentinel, and Terraform state file protection. We analyze common misconfiguration patterns observed in real-world Terraform codebases comprising 340 modules and 1,200 resource definitions across AWS and Azure environments, and demonstrate how systematic automated scanning identified 290 security misconfigurations and enabled 94 percent automated remediation. The paper also addresses Ansible-specific security considerations including vault management, playbook injection risks, Ansible Galaxy supply chain concerns, and the comparative gap in static analysis tool coverage between Terraform and Ansible codebases. We conclude with practical recommendations for organizations integrating IaC security into their DevSecOps pipelines and discuss the evolving regulatory landscape driving IaC compliance requirements.

Keywords - Infrastructure as Code, Terraform, Ansible, Security, Static Analysis, Checkov, TFSEC, Open Policy Agent, Cloud Misconfiguration, Policy as Code, Terraform State, Ansible Vault, DevSecOps.

1. Introduction

The promise of Infrastructure as Code is compelling: treat infrastructure the same way we treat application code version-controlled, peer-reviewed, tested, and deployed through automated pipelines. Organizations adopting IaC report faster provisioning times, reduced configuration drift, and improved reproducibility across environments. The Terraform State of Infrastructure Report (2025) indicates that over 80 percent of Fortune 500 companies use Terraform for at least some cloud infrastructure provisioning, and Ansible remains the most popular configuration management tool in the DevOps ecosystem.

But there is a fundamental tension in this model that few organizations recognize until they experience it firsthand. The same automation that makes IaC powerful also makes it dangerous when misconfigured. In manual infrastructure management, a misconfigured security group affects one resource. In IaC, a misconfigured module used across 50 deployments creates 50 misconfigured resources simultaneously. The blast radius is proportional to the reuse of the code.

I have personally investigated incidents where a Terraform module that created S3 buckets was updated to remove the Block Public Access setting and within 20 minutes, 14 buckets across three AWS accounts became publicly accessible. The change passed code review because the reviewer focused on the feature being added (lifecycle policies) and did not notice the security setting that was removed. This is not a failure of individual competence it is a systemic problem that requires automated tooling to address.

Industry research consistently shows that cloud security incidents are overwhelmingly caused by misconfigurations rather than sophisticated exploits. The 2024 Verizon Data Breach Investigations Report reported that misconfiguration was the leading cause of cloud data breaches for the fourth consecutive year. Publicly accessible storage buckets, overly permissive security groups, unencrypted databases, storage accounts without access logging, IAM policies with wildcard permissions these are exactly the types of problems that IaC security tooling can detect and prevent before they reach production.

This work builds on our earlier research into cloud migration patterns [1] and disaster recovery automation [2] by addressing the security dimension of the IaC workflows that underpin both domains. The framework presented here is informed by three years of operational experience managing Terraform and Ansible codebases for enterprise telecommunications infrastructure.

1.1. Research Contributions

This paper makes the following contributions:

- A four-layer IaC security framework (pre-commit, CI pipeline, policy-as-code, state protection) with practical implementation details for each layer
- Empirical analysis of 290 misconfigurations discovered across 340 Terraform modules in production AWS and Azure environments
- Comparative evaluation of IaC scanning tools (Checkov, tfsec, KICS, Terrascan) across detection coverage, performance, and usability dimensions
- Identification and analysis of Ansible-specific security risks that existing tools inadequately address
- Practical remediation timeline and approach for organizations with existing IaC codebases

2. Background and Related Work

2.1. Terraform Security Model

Terraform uses a declarative configuration language (HCL HashiCorp Configuration Language) to define infrastructure resources across cloud providers. Unlike imperative tools that specify step-by-step instructions, Terraform describes the desired end state and calculates the operations needed to achieve it. This declarative model simplifies infrastructure management but creates several security concerns that span multiple layers.

The first concern is the Terraform state file. Terraform maintains a JSON state file that records the current state of all managed resources, including their full attribute sets. This state file frequently contains sensitive data: database passwords (if defined inline), private IP addresses, certificate private keys (for tls_private_key resources), and API endpoints. If the state file is stored in a local directory, version-controlled in Git, or stored in an unencrypted remote backend, this sensitive data is exposed.

The second concern is provider credential management. Terraform providers (AWS, Azure, GCP, Kubernetes) require authentication credentials that grant broad infrastructure modification capabilities. These credentials are commonly passed through environment variables, shared credential files, or in the worst case hardcoded in Terraform configuration files. A compromised Terraform execution environment inherits whatever cloud permissions the provider credentials grant.

The third concern is module supply chain integrity. Terraform modules can be sourced from the public Terraform Registry, private registries, Git repositories, or local paths. Public modules are community-maintained and may contain insecure defaults, known vulnerabilities, or even malicious code. Unlike application package managers (npm, pip), the Terraform Registry has limited automated security scanning of published modules.

The fourth and most prevalent concern is resource misconfiguration the actual security settings defined in Terraform code that determine whether resources are encrypted, access-controlled, logged, and properly networked. This is the domain where static analysis tools provide the most value.

2.2. Ansible Security Considerations

Ansible operates as an agentless configuration management tool using SSH for Linux targets and WinRM for Windows targets. Because Ansible executes commands on remote systems, its security model is fundamentally about controlling what runs, who can trigger it, and how secrets are managed during execution.

Ansible Vault provides symmetric AES-256 encryption for secrets stored alongside playbooks. Variables, files, or entire YAML files can be encrypted with a vault password. The challenge is vault password management itself the password must be available at execution time, which means it is typically stored in a file, passed as an environment variable, or retrieved from a secret management system. If the vault password is committed to version control (a surprisingly common mistake), the encrypted vault files provide no protection.

Playbook injection is a risk when Ansible variables accept user-supplied input that is interpolated into shell commands or template files. Ansible's template engine (Jinja2) renders variables directly into templates without automatic escaping, which means that a variable containing shell metacharacters can inject arbitrary commands when used in a shell or command task.

Ansible Galaxy, the public repository for Ansible roles and collections, presents a supply chain risk analogous to npm or PyPI. Community roles may contain insecure defaults (installing services that listen on 0.0.0.0, using passwordless sudo, disabling SELinux), outdated dependencies, or intentionally malicious code. Unlike Terraform modules, Ansible Galaxy has minimal automated security review of published content.

2.3. Static Analysis Tool Landscape

Several purpose-built tools perform static security analysis of IaC code. The landscape has matured significantly over the past three years, with tools now covering not just Terraform but also CloudFormation, Kubernetes manifests, Dockerfiles, and Ansible playbooks.

Table 1. Comparison of Infrastructure-as-Code Security and Configuration Scanning Tools

Tool	Developer	Terraform	Ansible	K8s / Docker	Policy Count	Custom Rules
Checkov	Palo Alto / Bridgecrew	Excellent	Basic	Good	1,000+	Python / YAML
tfsec	Aqua Security	Excellent	No	No	500+	JSON / HCL
KICS	Checkmarx	Good	Good	Good	800+	Rego

Terrascan	Tenable	Good	Partial	Good	700+	Rego
Trivy (IaC)	Aqua Security	Good	No	Good	600+	Rego
ansible-lint	Community	No	Excellent	No	100+	Python

A critical observation is the coverage gap for Ansible. While Terraform has excellent coverage from multiple overlapping tools, Ansible security scanning is significantly less mature. Checkov includes some Ansible checks (primarily for AWS-related tasks), but coverage is incomplete. ansible-lint focuses primarily on code quality and best practices rather than security misconfigurations. This gap is concerning given Ansible's widespread use in production configuration management.

2.4. Regulatory Drivers

Regulatory frameworks increasingly expect automated infrastructure compliance verification. SOC 2 Type II auditors ask for evidence of configuration management controls. PCI-DSS 4.0 requires automated detection of security misconfigurations. The EU Cyber Resilience Act will mandate documented security processes for all products with digital elements by 2027. IaC security scanning provides continuous, machine-verifiable evidence of compliance that auditors can review replacing manual evidence collection spreadsheets with automated dashboard reporting.

2.5. Common Terraform Misconfiguration Patterns

Before describing the scanning framework, it is useful to understand the specific misconfiguration patterns that appear most frequently in production Terraform codebases. The patterns below are drawn from our analysis of 340 Terraform modules and corroborated by industry reports from Palo Alto's Unit 42 Cloud Threat Report and Datadog's State of Cloud Security.

2.5.1. Overly Permissive Network Rules

The most common and dangerous pattern is security groups or network security groups (NSGs) with 0.0.0.0/0 ingress rules on sensitive ports. Developers add these during development for convenience (to access a test database from their local machine) and forget to restrict them before merging. In one incident we investigated, a development RDS instance with a 0.0.0.0/0 ingress rule on port 5432 was accessible from the internet for 47 days before a cloud security scan detected it. The instance contained test data that was a sanitized copy of production data but the sanitization was incomplete, and some customer email addresses were exposed.

```
# BAD: Open to the world
resource "aws_security_group_rule" "allow_postgres" {
  type              = "ingress"
  from_port         = 5432
  to_port           = 5432
  protocol          = "tcp"
  cidr_blocks       = ["0.0.0.0/0"]   # DANGER
  security_group_id = aws_security_group.db.id
}
```

```
# GOOD: Restricted to application subnet
resource "aws_security_group_rule" "allow_postgres" {
  type              = "ingress"
  from_port         = 5432
  to_port           = 5432
  protocol          = "tcp"
  cidr_blocks       = ["10.0.4.0/24"]  # app subnet only
  security_group_id = aws_security_group.db.id
}
```

2.5.2 Missing Encryption Configuration

The second most common pattern is resources created without encryption at rest. In AWS, S3 buckets now encrypt by default (since January 2023), but this only applies to new buckets created through the console Terraform provisions with whatever the configuration specifies. If the encryption block is omitted in Terraform code written before the default encryption change, the bucket may not have server-side encryption, or it may use S3-managed keys (SSE-S3) instead of the customer-managed KMS keys required by organizational policy.

Similarly, RDS instances, EBS volumes, Azure Storage accounts, and Azure SQL databases all support encryption at rest but do not always enable it by default in Terraform configurations. The fix is straightforward add the encryption configuration block but the challenge is identifying all 47 resources across 340 modules that are missing it.

2.5.3 Public Access on Storage Resources

S3 bucket public access and Azure Blob Storage anonymous access are the cloud security equivalent of an unlocked front door. AWS has made S3 block public access the default for new buckets since April 2023, but Terraform modules created before that date and modules using older AWS provider versions may not enforce this. The correct Terraform configuration requires an explicit aws_s3_bucket_public_access_block resource with all four settings (block_public_acls, block_public_policy, ignore_public_acls, restrict_public_buckets) set to true.

2.5.4 Hardcoded Secrets and Credentials

Despite every security training program warning against it, hardcoded secrets appear in Terraform code with alarming frequency. The most common patterns are: database master passwords defined as default values in Terraform variables, API keys in provider blocks, SSH private keys in provisioner blocks, and TLS certificates in template literals. The dangerous subtlety is that even if the secret is defined in a terraform.tfvars file that is gitignored, it still appears in plaintext in the Terraform state file.

The correct approach is to never define secrets in Terraform code at all. Use dynamic secrets from HashiCorp

Vault, AWS Secrets Manager, or Azure Key Vault via the respective Terraform provider data sources. For bootstrap scenarios where a Vault integration is not yet available, use Terraform's sensitive variable marking (Terraform 0.14+) and the random_password resource, ensuring the state file is encrypted.

3. Proposed Security Framework

3.1. Layer 1: Pre-Commit Scanning

The first layer of defense operates in the developer's local environment before code reaches version control. We configure pre-commit hooks using the pre-commit framework that run tfsec and Checkov on staged Terraform files, blocking commits that contain critical or high-severity findings.

The pre-commit approach catches the most obvious misconfigurations open security groups, missing encryption, public access enabled before code review. This is the cheapest point in the development lifecycle to fix issues. A developer who receives immediate feedback that their security group allows 0.0.0.0/0 ingress can fix it in seconds. The same finding discovered during a production audit may require a change request, impact assessment, downtime window, and post-change verification.

```
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/antonbabenko/pre-commit-tf
    rev: v1.86.0
    hooks:
      - id: terraform_fmt
      - id: terraform_validate
      - id: terraform_tfsec
        args: ['--minimum-severity=HIGH']
      - id: terraform_checkov
        args: ['--check', 'CKV_AWS,CKV_AZURE,CKV2']
```

A practical consideration: pre-commit hooks should run fast. If the scan takes more than 10-15 seconds, developers will start bypassing it with --no-verify. We configure pre-commit to scan only staged files (not the entire repository) and to skip informational-only findings, keeping the typical execution time under 5 seconds.

An important nuance about pre-commit scanning: it catches misconfigurations in the raw HCL source code, but it cannot catch issues that only manifest in the computed Terraform plan. For example, a module might have a variable with a default value of public = true that the caller does not override. The source code shows the caller's configuration (which does not mention the public variable at all), but the plan shows the actual value that will be applied. Pre-commit scanning will miss this only plan-level scanning catches it.

Despite this limitation, pre-commit scanning is valuable because it catches the most common and most egregious misconfigurations: explicit 0.0.0.0/0 rules, missing encryption blocks, hardcoded credentials, and resources without tags. These represent approximately 70 percent of the findings we see in practice.

3.2. Layer 2: CI Pipeline Integration

The second layer runs comprehensive IaC analysis in the CI pipeline. This layer is mandatory developers can skip pre-commit hooks, but they cannot skip the CI pipeline. The pipeline performs two distinct types of scanning:

3.2.1 Source Code Scanning

Source code scanning analyzes the raw HCL files to detect patterns that match known insecure configurations. This catches misconfigurations that are explicitly stated in the code for example, a security group rule with cidr_blocks = ["0.0.0.0/0"] or an S3 bucket with no server_side_encryption_configuration block.

3.2.2 Plan-Level Scanning

Plan scanning is significantly more powerful than source scanning because it evaluates the computed state that Terraform will apply. This catches misconfigurations that source scanning cannot detect: default values from modules (a module may set public_access = true by default if the caller does not explicitly set it to false), computed values from data sources, and attribute inheritance across resources.

We generate a Terraform plan in JSON format and scan it with both Checkov and Conftest (OPA runner for structured data):

```
# CI Pipeline: Terraform security scanning
stage('Terraform Init') {
  steps {
    sh 'terraform init -backend=false'
  }
}
stage('Terraform Plan') {
  steps {
    sh 'terraform plan -out=tfplan'
    sh 'terraform show -json tfplan > plan.json'
  }
}
stage('Source Scan — tfsec') {
  steps {
    sh 'tfsec . --format sarif --out tfsec-results.sarif --minimum-severity HIGH'
    recordIssues(tools: [sarif(pattern: 'tfsec-results.sarif')])
  }
}
stage('Plan Scan — Checkov') {
  steps {
    sh 'checkov -f plan.json --framework terraform_plan --soft-fail'
  }
}
stage('Policy Scan — Conftest') {
  steps {
```

```
    sh 'conftest test plan.json --policy ./policies/ --no-fail'
  }
}
```

Notice that both Checkov and Conftest run in --soft-fail or --no-fail mode. This is intentional for the initial rollout period. During the first month, we run scans in advisory mode findings are reported but do not block the pipeline. This allows teams to see the volume and nature of findings without being blocked on every pull request. After the initial remediation phase, we switch to enforcement mode where critical and high findings block the pipeline.

3.3. Layer 3: Policy-as-Code Guardrails

Beyond scanning for known bad patterns, we implement organizational policies using Open Policy Agent (OPA) with Rego rules. These policies enforce business-specific requirements that generic scanning tools do not cover. While Checkov can detect 'missing encryption,' it cannot enforce 'encryption must use the KMS key from the security-keys account' that is an organizational policy.

Our standard OPA policy set includes:

- All S3 buckets must have versioning enabled and server-side encryption with customer-managed KMS keys (not AWS-managed SSE-S3)
- Security groups must not contain 0.0.0.0/0 ingress rules except for port 443 on ALB listener security groups
- All RDS instances must have automated backups enabled with a minimum 7-day retention period and encryption at rest
- All EC2 instances must use approved AMI IDs published by the internal golden image pipeline — no marketplace or public AMIs
- All resources must carry five mandatory tags: Owner, Environment, CostCenter, DataClassification, and ManagedBy
- No IAM policies with Action: * or Resource: * (wildcard permissions) except in the security-admin account
- All Terraform remote state backends must use encryption and DynamoDB locking

Here is an example Rego policy enforcing KMS encryption on S3 buckets:

```
# policy/s3_encryption.rego
package terraform.s3

deny[msg] {
  resource := input.resource_changes[_]
  resource.type == "aws_s3_bucket"
  not has_kms_encryption(resource)
  msg := sprintf(
    "S3 bucket %s must have SSE-KMS encryption with customer-managed key [Policy: ORG-S3-001]",
    [resource.address]
  )
}

has_kms_encryption(resource) {
  enc := resource.change.after.server_side_encryption_configuration[_]
  rule := enc.rule[_]
  apply := rule.apply_server_side_encryption_by_default[_]
  apply.sse_algorithm == "aws:kms"
  apply.kms_master_key_id != null
}
```

3.4. Layer 4: Terraform State Security

The Terraform state file deserves its own layer of protection because it is the single most sensitive artifact in the Terraform workflow. Let me be explicit about what the state file contains: every attribute of every managed resource, including computed attributes that may never appear in the Terraform configuration files themselves.

For example, if you create an RDS instance with Terraform and specify the master password inline (a practice we prohibit but that exists in legacy code), that password appears in plaintext in the state file. Even if you subsequently change the password through the AWS console, the original password remains in the state file history. If you use the tls_private_key resource to generate a TLS certificate, the private key is stored in the state file. If you create a random_password resource, the generated password is in the state file.

Our state protection requirements include:

- Encrypted Storage: Remote state backend: S3 with SSE-KMS encryption and bucket versioning for AWS; Azure Blob with CMK encryption for Azure
- State Locking: DynamoDB table for AWS; Azure Blob lease for Azure prevents concurrent modifications that could corrupt state
- Access Restriction: S3 bucket policy restricts access to the CI/CD pipeline IAM role and the platform engineering team. No developer has direct state access.
- State Backup: S3 versioning + 90-day retention enables recovery from state corruption or accidental deletion
- Audit Logging: S3 access logging + CloudTrail data events provide full audit trail of state file access

```
# backend.tf — Secure state backend configuration
terraform {
  backend "s3" {
    bucket         = "myorg-terraform-state-prod"
    key            = "network/vpc/terraform.tfstate"
    region         = "ca-central-1"
    encrypt        = true
    kms_key_id     = "arn:aws:kms:ca-central-1:123456789:key/abc-def-ghi"
    dynamodb_table = "terraform-state-lock"
  }
}
```

3.5. Ansible Security Controls

While the previous sections focused on Terraform, organizations using Ansible for configuration management need parallel controls. Our Ansible security practices include:

- Vault Management: All secrets encrypted with Ansible Vault using AES-256. Vault passwords retrieved from HashiCorp Vault at runtime never stored in files or environment variables.
- Input Validation: All user-supplied variables validated with assert tasks before use in shell commands. Jinja2 templates use the | quote filter for values interpolated into shell contexts.
- Galaxy Supply Chain: Community roles pinned to specific versions (not 'latest'). All roles reviewed before first use. Internal roles preferred over Ansible Galaxy.
- Linting: ansible-lint with custom rules enforced in CI pipeline. Focus areas: no bare variables in shell tasks, no use of raw module, no use of command/shell when a native module exists.

3.5.1. Ansible Vault Best Practices in Detail

Ansible Vault encryption is straightforward in principle encrypt sensitive variables or files with a password but the practical implementation has several subtle pitfalls that many teams discover only after a security incident.

The first pitfall is vault password storage. The vault password itself must be available at playbook execution time, which creates a chicken-and-egg problem: the password that protects your secrets must itself be stored securely. We solve this by retrieving the vault password from HashiCorp Vault (the secrets management tool, not Ansible Vault the feature) at the start of the CI pipeline via a vault-password-client script that Ansible invokes automatically.

```
# vault-password-client.sh — retrieves vault password from HashiCorp Vault
#!/bin/bash
set -euo pipefail
# VAULT_ADDR and VAULT_TOKEN are set by CI pipeline environment
vault kv get -field=ansible_vault_password secret/ci/ansible
```

The second pitfall is vault key rotation. Ansible Vault uses a single symmetric key. If a team member who knows the vault password leaves the organization, the password should be rotated and all encrypted files re-encrypted. The ansible-vault rekey command makes this operationally feasible, but it requires touching every encrypted file in the repository which can be hundreds of files in a large Ansible codebase.

The third pitfall is partial encryption. When developers encrypt individual variables within a YAML file (encrypted string values), the variable names remain in plaintext. A variable named database_admin_password: !vault | ... reveals the existence (though not the value) of a database admin credential, which provides information to an attacker about the infrastructure topology.

3.5.2. Playbook Injection Prevention

Ansible's power comes from its ability to execute arbitrary commands on remote systems. This same power makes it dangerous when variables containing untrusted input are interpolated into command tasks without proper escaping.

Consider a playbook that creates system users based on a variable list:

```
# VULNERABLE: Variable directly in shell command
- name: Create user accounts
  shell: "useradd -m {{ item.username }}"
  loop: "{{ users }}"

# If item.username = 'alice; rm -rf /', this executes:
# useradd -m alice; rm -rf /

# SAFE: Using the user module (no shell interpretation)
- name: Create user accounts
  user:
    name: "{{ item.username }}"
    state: present
    create_home: true
  loop: "{{ users }}"
```

The safe approach uses Ansible's native user module rather than a shell command. The native module handles input safely without shell interpretation. Our ansible-lint custom rules flag any use of shell or command modules where a native module exists for the same operation, and require the quote Jinja2 filter when shell module usage is genuinely necessary (for operations without a native module equivalent).

3.5.3. Ansible Galaxy Supply Chain Mitigation

We manage Ansible Galaxy supply chain risk through a four-step process:

- All Galaxy roles and collections are downloaded into an internal mirror repository. Teams install from the mirror, not directly from Galaxy.
- New roles undergo a security review before being added to the mirror. The review checks for: shell commands with unquoted variables, use of become: yes, without restriction, services binding to 0.0.0.0, disabled security features (SELinux, firewall rules), and hardcoded credentials.
- All roles are pinned to specific version numbers in requirements.yml. The version pin includes both the Galaxy version and a commit SHA for Git-sourced roles.
- A monthly review process checks for known vulnerabilities in pinned role versions and updates as needed. This is manual today we are exploring automation using Dependabot-style tools for Ansible requirements files.

3.6. End-to-End Secure IaC Workflow

To illustrate how the four layers work together, consider a developer making a change to a Terraform module that provisions an RDS database instance. The workflow proceeds through the following steps:

Step 1 — Local Development. The developer modifies the RDS module to add a new read replica. They run terraform fmt and terraform validate locally to ensure syntax correctness. When they attempt to commit, the pre-commit hook runs tfsec on the staged files. tfsec detects that the new read replica does not have storage_encrypted = true. The developer adds the encryption configuration and commits successfully.

Step 2 — Pull Request. The developer pushes the branch and opens a pull request. The CI pipeline triggers automatically. It runs terraform init and terraform plan to generate the execution plan, then scans the plan with Checkov. Checkov identifies that the read replica's backup_retention_period is set to 0 (no automated backups). The developer is notified through a GitHub comment with the specific finding and remediation guidance.

Step 3 — Policy Check. The CI pipeline also runs Conftest with the organizational OPA policies. A policy validates that the RDS KMS key ARN matches the approved encryption key for the target account and region. The developer had used the wrong KMS key (a key from the development account for a production resource). Conftest fails the policy check. The developer corrects the KMS key reference and pushes the fix.

Step 4 — Apply. After the pull request passes all scanning layers and receives peer review approval, it is merged. The deployment pipeline runs terraform apply. The new read replica is created with encryption enabled, automated backups configured, and the correct KMS key. The state file is updated in the encrypted S3 backend with DynamoDB locking.

Without the scanning framework, this workflow would have produced a read replica without encryption, without backups, and with the wrong encryption key three separate misconfigurations from a single routine change. Each of these would have been caught eventually by a quarterly security audit, but they would have been in production for weeks or months before detection.

4. Results

4.1. Misconfiguration Discovery

We applied the scanning framework to a production Terraform codebase comprising 340 modules and 1,200 resource definitions across AWS (ca-central-1, us-east-1) and Azure (Canada Central) environments. The codebase had been in active development for approximately two years before scanning was introduced. The results were sobering.

Table 2. Cloud Infrastructure Misconfiguration Analysis: Severity Breakdown and Auto-Fixability Assessment

Misconfiguration Category	Count	Critical	High	Medium	Auto-Fixable
Missing encryption at rest (S3, RDS, EBS, Azure Storage)	47	12	35	0	41 (87%)
Overly permissive security groups (0.0.0.0/0 ingress)	32	8	24	0	28 (88%)
Public access on storage resources (S3, Azure Blob)	18	18	0	0	18 (100%)
Missing access logging (S3, ALB, CloudTrail gaps)	29	0	29	0	29 (100%)
Missing or incomplete resource tags	156	0	0	156	156 (100%)
IAM wildcard policies (Action: * or Resource: *)	11	11	0	0	4 (36%)
Secrets in plaintext Terraform variables	8	8	0	0	0 (Vault migration)
Unencrypted Terraform state backends	3	3	0	0	3 (100%)
Total	304	60	88	156	279 (92%)

4.2. Remediation Timeline

We prioritized remediation by severity. Critical findings (60 items) were addressed in the first sprint (two weeks), with the security team directly fixing public access and plaintext secrets. High findings (88 items) were assigned to the owning teams and completed over the next two sprints. Medium findings (tags) were addressed through a bulk Terraform refactoring effort in the fourth sprint.

The most complex remediation was the eight plaintext secret references, which required migration from inline Terraform variables to HashiCorp Vault dynamic secrets with a Terraform Vault provider integration. This took an additional sprint to complete because it required changes to the CI/CD pipeline, Vault policy configuration, and application deployment workflows.

After the initial remediation, ongoing CI pipeline scanning maintained a zero-critical-finding state for the subsequent six months. New findings averaged 3-4 per week (mostly medium-severity tag compliance issues on newly created resources), all caught and remediated before reaching production.

4.3. Tool Performance Comparison

We ran all four scanning tools against the same Terraform codebase to compare detection coverage and performance:

Table 3. Comparison of IaC Security Scanners: Detection Coverage, Accuracy, and Performance Analysis

Tool	Findings (Critical+High)	Findings (Medium)	False Positives	Scan Time	Unique Findings
Checkov	142	156	8 (5%)	45 seconds	12 — unique module-level checks
tfsec	138	89	3 (2%)	12 seconds	6 — deep HCL parsing finds
KICS	128	134	14 (10%)	38 seconds	4 — cross-resource correlation
Terrascan	119	98	11 (9%)	28 seconds	2 — Rego-based custom detection

The key finding is that no single tool catches everything. Checkov found 12 unique findings that other tools missed (primarily around Terraform module default values). tfsec found 6 unique findings through deeper HCL parsing. Running Checkov + tfsec together provides the highest coverage with the lowest false positive rate, which is our recommended combination.

4.4. Real-World Case Study: Telecommunications Terraform Codebase

To provide concrete context, our telecommunications infrastructure is managed through 340 Terraform modules organized in a monorepo structure. The modules cover VPC/VNet networking (42 modules), compute instances and auto-scaling (38 modules), RDS and Azure SQL databases (27 modules), S3 and Azure Blob storage (31 modules), IAM roles and policies (45 modules), Kubernetes cluster provisioning (22 modules), monitoring and logging infrastructure (35 modules), DNS and CDN configuration (18 modules), and shared library modules providing common patterns (82 modules).

The codebase had evolved over approximately two and a half years, written by 14 engineers with varying levels of security awareness. Early modules were created during the initial cloud migration when the priority was functionality over security a common and understandable pattern that accumulates security debt over time. The scanning framework was introduced as part of a broader DevSecOps maturation initiative.

The most impactful finding category was the 11 IAM wildcard policies IAM policies with Action: "*" or Resource: "*" that granted far more permissions than the workloads needed. These had been created during initial development when engineers did not know the exact permissions required and used wildcards as a shortcut. Remediating these required working with each team to understand the actual API calls their applications made (using CloudTrail analysis) and replacing the wildcard policies with minimum-privilege policies. This was the most time-intensive remediation category, taking approximately three sprints of dedicated work.

4.4.1. The S3 Public Access Incident

The scanning framework identified 18 S3 buckets that were missing the aws_s3_bucket_public_access_block resource. Of these, 14 were internal-only buckets that had never been configured for public access (but lacked the explicit block that would prevent future misconfiguration), and 4 were actively misconfigured with public read access. Two of the four were development environment buckets containing non-sensitive test data. The other two contained application logs from a production API service not customer data, but operational data that could reveal internal architecture information to an external observer.

These two buckets had been publicly accessible for approximately seven months before the scan detected them. The root cause was a Terraform module that created S3 buckets without a public access block, and the specific module invocation had acl = "public-read" set apparently copied from a tutorial without understanding the implications. The remediation was immediate: add the public access block and change the ACL to private. But the incident underscored the need for automated scanning seven months of exposure that would have been prevented by a pre-commit check.

4.4.2. The Plaintext Secrets Migration

The eight plaintext secret references were the most complex remediation. They included four RDS master passwords defined as Terraform variables with default values, two API keys for third-party services hardcoded in Terraform data source filters, one SSH private key in a provisioner block, and one TLS private key generated by the tls_private_key resource (which is inherently stored in state).

The migration path was different for each category. RDS passwords were migrated to AWS Secrets Manager with the Terraform AWS provider's secretsmanager_secret_version resource, then rotated immediately. The API keys were moved to HashiCorp Vault and retrieved via the Vault provider's vault_generic_secret data source. The SSH key was replaced with SSM Session Manager access (eliminating SSH entirely). The TLS private key was migrated to AWS Certificate Manager, removing the need for Terraform to manage the key at all.

After migration, we implemented a Checkov custom check that scans for common secret patterns in Terraform code (passwords, api_key, secret_key, private_key as variable names with default values) and blocks any new introduction of plaintext secrets.

4.5. Before and After Metrics

We tracked several metrics to measure the impact of the security framework over the six months following full deployment.

Table 4. Security Transformation Metrics After Framework Implementation

Metric	Before Framework	After Framework (6 months)	Change
Critical security findings in production	60	0	-100%
High severity findings in production	88	3 (all with documented exceptions)	-97%
Mean time to detect misconfiguration	Unknown (found during audits)	< 5 minutes (CI pipeline)	Dramatic improvement
Mean time to remediate misconfiguration	2-4 weeks (audit cycle)	< 1 hour (developer self-service)	~99% reduction
New findings per week (steady state)	N/A	3-4 (medium severity, tags)	Manageable volume
Developer deployment pipeline time increase	N/A	+90 seconds (scan time)	Minimal impact
Security audit evidence preparation time	2-3 days per quarter	Automated (continuous)	~100% reduction

5. Discussion

5.1. Shift-Left Economics

The results reinforce a principle familiar from application security: the earlier a misconfiguration is detected, the cheaper it is to fix. Pre-commit scanning catches issues in seconds at zero operational cost the developer fixes the problem before it ever leaves their machine. The same misconfiguration discovered in a production audit may require a change management ticket, risk assessment, scheduled maintenance window, infrastructure modification affecting running workloads, and post-change verification a process that typically takes 2-4 weeks and involves three or more teams.

We estimate the fully loaded cost of remediating a misconfiguration at each stage:

Table 5. Cost, Remediation Time, and Risk Exposure Across Security Detection Stages

Detection Stage	Estimated Cost per Finding	Time to Remediate	Risk Window
Pre-commit (developer workstation)	$0 (developer self-service)	< 5 minutes	0 (never deployed)
CI pipeline (pull request)	$50 (developer + reviewer time)	< 1 hour	0 (not merged)
Staging environment scan	$200 (cross-team coordination)	1-3 days	Days (staging exposure)
Production audit / breach	$5,000-50,000+ (incident response)	1-4 weeks	Weeks to months

5.2. Ansible Coverage Gap

A notable limitation of the current tool landscape is Ansible coverage. While Terraform scanning tools are mature, overlapping, and comprehensive, Ansible security scanning lags significantly behind. Checkov includes approximately 50 Ansible checks (primarily for AWS-related tasks), while Terraform has over 1,000. ansible-lint provides excellent code quality and best practice enforcement but lacks security-specific rules for common misconfigurations like insecure sudo configurations, disabled SELinux, overly permissive file modes, and unvalidated variable interpolation in command tasks.

Organizations relying heavily on Ansible should supplement automated scanning with custom Rego policies applied to playbook YAML (using Conftest), manual security review of all Galaxy roles before adoption, and ansible-lint custom rules targeting their specific security requirements.

5.3. Organizational Adoption Challenges

The technical implementation of IaC scanning is straightforward. The organizational adoption is harder. Common resistance patterns we observed include: 'These are false positives' (sometimes true tool tuning is required), 'Security is slowing us down' (the 5-second pre-commit scan is not the bottleneck), and 'We will fix it later' (which never happens without pipeline enforcement). Successful adoption requires executive sponsorship, a phased rollout (advisory mode first, enforcement second), and clear documentation of exception processes for legitimate false positives.

The phased rollout approach deserves elaboration because it was critical to our success. In month one, we deployed scanning in advisory mode findings were surfaced as PR comments but did not block merges. This visibility phase served two purposes: it gave teams time to understand the findings and begin remediation, and it gave the security team data to tune the scanning tools and suppress confirmed false positives. Without this tuning phase, teams would have been blocked by false positives on day one, destroying trust in the tooling.

In month two, we switched to 'soft enforcement' critical findings blocked merges, but high and medium findings remained advisory. This was the minimum viable enforcement level that prevented the most dangerous misconfigurations (public storage, plaintext secrets, wildcard IAM policies) from reaching production while still allowing teams time to address less critical findings.

In month three, we moved to full enforcement critical and high findings blocked merges. Medium findings remained advisory because the volume (mostly tag compliance) was too high to enforce immediately. By month four, tag remediation was complete and medium enforcement

was enabled. The entire transition from zero scanning to full enforcement took four months, which felt slow at the time but was essential for organizational buy-in.

5.4. Compliance Mapping: IaC Scanning to Regulatory Frameworks

A particularly valuable aspect of IaC scanning is its ability to provide continuous, machine-verifiable evidence for compliance audits. Organizations subject to SOC 2, PCI-DSS, HIPAA, or NIST 800-53 spend significant effort preparing evidence packages for auditors. IaC scanning can automate much of this evidence collection.

Table 6. Mapping Compliance Requirements to IaC Security Controls and Automated Evidence Generation

Compliance Requirement	Framework	IaC Scanning Control	Evidence Generated
Encryption at rest for stored data	SOC 2 CC6.1 / PCI-DSS 3.4	Checkov check: S3/RDS/EBS encryption enabled	Continuous scan report showing all storage resources encrypted
Network segmentation	PCI-DSS 1.3 / NIST SC-7	tfsec check: no 0.0.0.0/0 on sensitive ports	Security group audit showing restricted ingress rules
Access control enforcement	SOC 2 CC6.3 / NIST AC-3	OPA policy: no wildcard IAM policies	Policy evaluation results showing least-privilege IAM
Configuration management	SOC 2 CC8.1 / NIST CM-2	CI pipeline scan on every infrastructure change	Pipeline execution logs showing scan pass/fail for every merge
Audit logging	SOC 2 CC7.2 / PCI-DSS 10.1	Checkov check: CloudTrail, S3 access logging enabled	Resource inventory showing logging enabled on all resources
Secrets management	SOC 2 CC6.7 / NIST IA-5	Custom Checkov check: no plaintext secrets in code	Scan results showing zero hardcoded credentials

During our most recent SOC 2 Type II audit, we provided the auditor with a dashboard showing six months of continuous scanning results, including every finding, its severity, when it was detected, and when it was remediated. The auditor noted that this was the most comprehensive configuration management evidence they had seen from any customer. In previous years, our audit evidence for the same controls consisted of manual screenshots taken the week before the audit a snapshot that told the auditor nothing about the state of controls during the rest of the audit period.

5.5. Multi-Cloud Considerations

Organizations operating across multiple cloud providers (AWS and Azure in our case) face additional IaC security challenges. Security controls are not identical across providers: AWS S3 and Azure Blob Storage have different default security configurations, different encryption options, and different access control models. A scanning rule that checks for server_side_encryption_configuration on aws_s3_bucket does not catch a missing encryption configuration on azurerm_storage_account.

This means that multi-cloud organizations need scanning rules for each provider. Checkov handles this well with provider-specific check prefixes (CKV_AWS_ for AWS, CKV_AZURE_ for Azure, CKV_GCP_ for GCP). But organizational OPA policies need to account for provider differences: the encryption policy for S3 buckets uses KMS key IDs, while the equivalent policy for Azure Storage accounts uses customer-managed keys from Azure Key Vault.

We maintain separate policy directories for each provider (policies/aws/, policies/azure/) with a shared policy library (policies/common/) for provider-agnostic rules like mandatory tags and naming conventions. The CI pipeline automatically selects the correct policy directory based on the Terraform provider blocks in the code being scanned.

5.6. Integration with Cloud Security Posture Management

IaC scanning is a preventive control that catches misconfigurations before deployment. Cloud Security Posture Management (CSPM) tools (AWS Config, Azure Defender for Cloud, Prisma Cloud) provide detective controls that catch misconfigurations in running infrastructure. The two approaches are complementary: IaC scanning prevents misconfigurations from being deployed, while CSPM catches misconfigurations introduced through manual changes (console clicks), API calls outside Terraform, and configuration drift.

In an ideal environment, every infrastructure change goes through Terraform, and IaC scanning catches everything. In reality, engineers occasionally make manual changes through the cloud console especially during incident response when speed matters more than process. CSPM provides the safety net for these situations, and Terraform's drift detection (terraform plan showing unexpected changes) highlights the discrepancy for subsequent reconciliation.

5.7. Terraform Module Supply Chain Security

The Terraform Registry hosts thousands of community modules that simplify infrastructure provisioning. Using a community module for VPC creation, RDS provisioning, or EKS cluster setup can save days of development time. However, community modules carry supply chain risks that organizations must evaluate.

A community module is third-party code that runs with your cloud credentials. It can create any resource, modify

any configuration, and exfiltrate any secret that Terraform has access to. A malicious module author could add a data source that reads your AWS account's IAM credentials and sends them to an external endpoint and this would execute during terraform plan, before any apply confirmation.

Our mitigation strategy includes: pinning all module versions to specific tags (never using 'latest'), forking critical community modules into our internal registry for code review and controlled updates, running Checkov and tfsec on community module source code before adoption, and maintaining a registry of approved modules that teams may use without additional review.

5.8. The Terraform State Exposure Problem

I want to emphasize the state file security issue because it is the most underappreciated risk in the Terraform ecosystem. Most teams understand that they should encrypt their state file and store it remotely. Fewer understand the full implications of what the state file contains.

Consider a typical Terraform deployment that creates an RDS instance, an S3 bucket, and a Lambda function. The state file will contain: the full connection string for the RDS instance (including endpoint, port, and database name), the bucket ARN and region, the Lambda function ARN and execution role ARN. For resources with sensitive attributes (aws_db_instance master_password, aws_iam_access_key secret), the values are stored in the state in plaintext Terraform marks them as sensitive in output but stores them unencrypted in the state JSON.

This means that anyone with read access to the state file has read access to every secret managed by Terraform. In our initial assessment, three Terraform state backends were storing state in S3 buckets without encryption and with overly permissive bucket policies that granted read access to all developers in the AWS account. This effectively gave every developer read access to every database password, API key, and TLS certificate managed by Terraform regardless of their RBAC permissions on the actual resources.

5.9. Future Directions

The IaC security tool landscape is evolving rapidly. Several trends will shape the next generation of IaC security practices:

- Auto-Remediation: AI-assisted remediation — tools like Snyk and Bridgecrew are beginning to offer automated fix suggestions that generate the correct Terraform code to remediate a finding, reducing the remediation burden on developers
- Native State Encryption: OpenTofu (the open-source Terraform fork) may introduce native state encryption at the tool level, addressing the state file exposure problem without requiring external encryption mechanisms
- Supply Chain Attestation: The SLSA framework (Supply-chain Levels for Software Artifacts) is being extended to IaC artifacts, providing a standardized way to verify the provenance and integrity of Terraform modules
- Alternative IaC Paradigms: Crossplane and Pulumi represent alternative IaC paradigms (Kubernetes-native and general-purpose programming languages, respectively) that may displace some Terraform use cases and require their own security scanning ecosystems

6. Conclusion

This paper presented a four-layer IaC security framework: pre-commit scanning for immediate developer feedback, CI pipeline analysis for enforced gate quality, policy-as-code guardrails for organizational compliance requirements, and state file protection for the most sensitive artifact in the Terraform workflow. Applied to a production Terraform codebase comprising 340 modules and 1,200 resources, the framework identified 304 security misconfigurations and enabled 92 percent automated remediation.

The key takeaway for practitioners is that IaC security scanning should be treated as a mandatory component of IaC pipelines, not an optional enhancement. The tools are mature, the false positive rates are manageable (2-10 percent depending on the tool), and the cost of catching a misconfiguration in the CI pipeline is orders of magnitude lower than catching it in a production audit or, worse, a breach. Organizations adopting Terraform and Ansible should implement the scanning framework presented here as part of their standard DevSecOps pipeline configuration.

Future work should focus on closing the Ansible security scanning gap, developing standardized policy libraries for common compliance frameworks (SOC 2, PCI-DSS, NIST 800-53) in OPA/Rego format, and evaluating the emerging category of AI-assisted misconfiguration remediation tools that can automatically generate fixes for detected findings.

References

[1] Chaudhary, B. S. (2025). Insights into Cloud Migration (Migration to Azure/AWS). IJCET, 16(1). https://doi.org/10.34218/IJCET_16_01_101

[2] Chaudhary, B. S. (2026). Designing Automated Disaster Recovery Strategies for Hybrid Cloud Environments in Critical Infrastructure. Zenodo. https://doi.org/10.13140/RG.2.2.12036.39048

[3] Chaudhary, B. S. (2026). Automating System Monitoring and Management: Achieving Significant Time Savings and Reducing Downtime. IJCSERD, 15(1). https://doi.org/10.5281/zenodo.19003772

[4] Chaudhary, B. S. (2026). Proactive Infrastructure Monitoring and Observability: Enhancing Critical System Reliability. ISCSITR - IJSRIT, 7(1), 1-33. https://doi.org/10.63397/ISCSITR-IJSRIT_2026_07_01_001

[5] Chaudhary, B. S. (2026). Zero-Trust Security Architecture for Containerized Microservices in

Enterprise Telecommunications Networks. Zenodo. https://doi.org/10.13140/RG.2.2.18747.27686

[6] Bridgecrew / Palo Alto Networks. "Checkov: Infrastructure as Code Static Analysis." checkov.io, 2025.

[7] Aqua Security. "tfsec: Security Scanner for Terraform." github.com/aquasecurity/tfsec, 2025.

[8] Checkmarx. "KICS: Keeping Infrastructure as Code Secure." kics.io, 2025.

[9] Open Policy Agent. "OPA: Policy-based control for cloud native environments." openpolicyagent.org, 2025.

[10] HashiCorp. "Terraform State: Purpose and Security." developer.hashicorp.com/terraform/language/state, 2025.

[11] Verizon. "2024 Data Breach Investigations Report." verizon.com/dbir, 2024.

[12] HashiCorp. "Sentinel Policy as Code Framework." hashicorp.com/sentinel, 2025.

[13] Ansible Documentation. "Ansible Vault." docs.ansible.com/ansible/latest/vault_guide/, 2025.

[14] Red Hat. "ansible-lint: Best Practices Checker for Ansible." ansible.readthedocs.io/projects/lint/, 2025.