Universal Library of Engineering Technology Research Article

ISSN: 3064-996X | Volume 2, Issue 3

Open Access | PP: 06-11

DOI: https://doi.org/10.70315/uloap.ulete.2025.0203002

Infrastructure Automation in Cloud Environments Using Terraform

Diyorjon Holkuziev

DevOps Engineer and Technical Lead at Brilom Inc., Boston, Massachusetts, United States.

Abstract

The article examines the principles and practices of automating the deployment and management of cloud infrastructure using Terraform. The relevance of this study is determined by the rapid increase in complexity of cloud environments and the risk of numerous incidents arising from manual configurations. Misconfigurations have become one of the leading causes of outages and data breaches. Accordingly, automating resource provisioning and management becomes a critical task to ensure reproducibility, security, and accelerated time-to-market. The objective of this work is to conduct a comprehensive analysis of Terraform's capabilities as a leading Infrastructure as Code tool for cloud-environment automation, to identify its advantages over alternative solutions (Pulumi, CloudFormation, Ansible), and to assess the impact of HashiCorp's latest features and services on infrastructure lifecycle management efficiency. The novelty of this research lies in an integrated review of Terraform's latest extensions: the introduction of provider-defined functions in version 1.8, the Stacks concept in HCP Terraform, background health assessments for drift detection, the module lifecycle management mechanism, and the HCP Terraform Premium plan uniting centralized migration, policy enforcement, and full change traceability. The methodological basis comprises comparative analysis, a systematic review of documentation, and content analysis of CI/CD integration practices. The main findings show that the declarative model and dry-run mechanism of Terraform ensure determinism of change and easier auditing; the unified state file and remote backends ensure consistency across all deployments. The wide provider ecosystem and modular architecture enable the rapid scaling of multi- and hybrid-cloud environments. Integration with GitHub Actions, Azure Pipelines, GitLab CI, and Bitbucket Pipelines via OIDC enhances security. New lifecycle management and drift-control capabilities transform Terraform into a full-fledged Infrastructure Lifecycle Management platform, reducing operational risk without cost escalation. This article will be useful for DevOps engineers, SREs, and cloud architects responsible for infrastructure automation and secure operations.

Keywords: *Terraform, Infrastructure as Code, Multi-Cloud Environments, Declarative Model, Drift Detection, CI/CD, OIDC.*

INTRODUCTION

Modern cloud environments are becoming increasingly complex, as a single enterprise system can encompass hundreds of services, dozens of regions, and multiple providers. In such a dynamic context, manual resource provisioning and maintenance not only impede time-to-market but also pose direct business risks. Gartner analysts predict that by the end of 2025, 99% of all incidents in public clouds will be caused by customer errors, primarily misconfigurations [1]. The financial stakes are corroborated by IBM's Cost of a Data Breach 2024 report: already, 40% of breaches involve data distributed across multiple environments, where control is particularly challenging [2].

The primary root of such losses is the lack of reproducibility. Every change made via console or CLI is introduced by a human operator, causing production state and documentation to diverge rapidly; this configuration drift complicates auditing and scaling. Infrastructure as Code (IaC) does so by specifying the target infrastructure in a format that is readable by machines. Furthermore, this approach promotes development practices within operations, including version control, code review, automated testing, and continuous integration.

Terraform is basically an Infrastructure as Code tool, and therefore the most popular. According to [3], Terraform leads 32.8% of IaC projects globally, thereby establishing itself as an uncontested leader in infrastructure automation solutions. Its popularity stems from a declarative model: engineers specify the desired outcome, while Terraform generates the execution plan across various clouds and on-premises platforms. A unified HCL language and over three thousand providers enable the description, within a single

Citation: Diyorjon Holkuziev, "Infrastructure Automation in Cloud Environments Using Terraform", Universal Library of Engineering Technology, 2025; 2(3): 06-11. DOI: https://doi.org/10.70315/uloap.ulete.2025.0203001.

repository, of an AWS VPC, an Azure Kubernetes cluster, and Google Cloud functions, all while maintaining a cohesive state in a distributed backend. Thus, Terraform not only automates mechanical tasks but also establishes a contract between development and operations teams, reducing the probability of errors to acceptable levels and supporting exponential growth of cloud workloads without a proportional increase in operational costs.

MATERIALS AND METHODOLOGY

This study is based on the analysis of 20 sources, including Gartner's forecast of public-cloud incidents caused by customer errors [1], IBM's report on data-compromise incidents in multi-environment settings [2], global Terraform usage statistics [3], an overview of provider-defined functions introduced in version 1.8 [4], materials on the Stacks concept in HCP Terraform [6], background health assessments for infrastructure drift detection [7], and the HCP Terraform Premium plan expanding lifecycle management capabilities [8].

The theoretical foundation comprises research elucidating the key principles of Infrastructure as Code and Terraform's role among automation tools: a comparative analysis of Pulumi, CloudFormation, and Terraform demonstrating Terraform's market share [3, 5]; a review of the declarative model and dry-run mechanism ensuring change determinism [9]; descriptions of the Terraform Registry's ecosystem capabilities [5]; and materials on modular resource lifecycle management and secure API refactoring [20].

Methodologically, the study integrates a comparative analysis of Infrastructure as Code tools—juxtaposing Terraform's declarative model [3, 5] with Pulumi's imperative approach and CloudFormation's ecosystem lock-in—a systematic review of HashiCorp's official documentation covering the core workflow, secure state and sensitive-data storage [9–11], and configuration and workspace management in Terraform Enterprise [12, 20], a content analysis of CI/CD practices—including Terraform integration into GitHub Actions via OIDC federation [14, 15], Azure Pipelines [16], GitLab CI [17], and Bitbucket Pipelines [18]—an evaluation of resource-drift control mechanisms such as background health assessments [7] and module lifecycle management [8], and an examination of DevSecOps approaches along with automated infrastructure validations [13].

RESULTS AND DISCUSSION

Terraform attracts attention primarily for its declarative model: an engineer describes the desired state, and the planner automatically computes the order of actions and verifies that the operations will bring the system exactly to that goal. This dry-run approach ensures determinism, allows peer review of forthcoming changes, and reduces the likelihood of errors in the operational lifecycle. With the release of version 1.8, the mechanism became even more flexible: configurations can now invoke provider-defined functions—extensions supplied by the provider itself and executed at planning time—removing limitations on embedding complex logic without external scripts [4].

A decisive factor remains the ecosystem. The public Terraform Registry hosts over 5,000 providers, covering not only AWS, Azure, and GCP but also SaaS platforms, network equipment, and on-premises solutions [5]. Thanks to this breadth, Terraform has secured a leading position in the global IaC tool market, significantly outpacing its nearest competitors. In multi-cloud scenarios, this advantage becomes critical: a single repository can describe infrastructure across multiple providers simultaneously, while a unified state file maintains consistency.

The platform's extensibility is not limited to functions. Users can package recurring patterns into modules, and providers can add their resources without needing to rebuild the core. Unlike AWS CloudFormation, which is tightly bound to a single ecosystem, and Pulumi, where infrastructure is described with imperative code, Terraform remains cloud-agnostic and requires minimal programming skills. Compared to Ansible, which is geared toward configuring existing instances, Terraform operates on the lifecycle of resources themselves and persists their state, which is crucial for large-scale cloud environments.

Over the past four years, the evolution of the 1.x series has aimed to narrow the gap between declarative intent and operational reality. In addition to provider-defined functions, versions 1.6–1.8 introduced a secure refactoring API and the native Terraform test command for writing unit tests for modules. The next step is the concept of Stacks. In HCP Terraform, a stack group related workspaces defines orchestration rules among them, and permits deferring the application of plan segments, thereby simplifying the coordination of complex releases [6].

In the managed HCP Terraform service, background health assessments now regularly compare actual infrastructure against the state file and signal resource drift caused by manual changes or cloud-provider issues [7]. The module lifecycle management feature provides an additional maturity layer, allowing administrators to mark module versions as deprecated or revoked, thereby forcing consumers to migrate to supported versions and preventing regressions [8].

Finally, the development trajectory is defined by the transition from simple declarative orchestration to full Infrastructure Lifecycle Management. In May 2025, HashiCorp introduced the HCP Terraform Premium plan, which includes automated migrations from the local CLI, centralized policies, and end-to-end change traceability—all designed to reduce operational risks when working with hybrid and multi-cloud architectures [8]. Terraform, hence, does not balance her stake as just a resource description language, but also as the

underpinning of a managed infrastructure change pipeline, wherein plan, validate, apply, and drift control are parts of a single chain.

The Terraform lifecycle is a deterministic sequence of actions: write → init → plan → apply → destroy. An engineer writes the desired infrastructure state and then runs Terraform init to download plugins and prepare the backend. The' terraform plan' command computes the change graph, showing which resources will be created, modified, or destroyed; this allows for review before any production intervention. After approval, Terraform applies the plan atomically, and for a full teardown, Terraform destroy is used. This model minimizes manual errors and ensures reproducible changes [9].

Reliable state management ensures the correctness of this cycle. The state file contains all attributes of managed resources, so storing it locally creates a single point of failure and a data-leak risk. It is recommended to offload state to a remote backend. HCP Terraform encrypts it at rest and secures it via TLS in transit, maintains a version history, and tracks change authors. Meanwhile, the S3 backend supports KMS encryption, object versioning, and DynamoDB-backed locking to prevent concurrent writes. All operations that may alter the state automatically acquire a lock to avoid corruption. HashiCorp Well-Architected Framework practices explicitly forbid storing state files in code repositories [10, 11].

For large projects, configurations are divided into modules. A module encapsulates a recurring resource pattern, accepts input variables, produces output values, and may reference local computations. This abstraction reduces code volume, simplifies testing, and enables versioned releases to public or private registries; Terraform grabs the version needed and locks it in a file. This way, you can count on consistent builds.

The structure of the repository affects the scalability of the process. HCP Terraform allows both—monorepo (more than one directory in a single repository) and polyrepo (independent repositories). In a monorepo, each workspace must specify its working directory and include shared modules in its file-trigger list; in a polyrepo, separate repositories enable releasing new versions of modules without updating all consumers at once. HashiCorp docs recommend using polyrepo unless there is a compelling reason for monolith because it speeds up CI processes and TEAM CONFLICT has less RISK [12].

A further safeguard is given by background drift in HCP Terraform which does this periodic comparison between the actual state of the cloud and the expected state and alerts if resources have been altered manually or as a result of service degradation.Detected drift can be exported in a report and addressed through the standard plan/apply cycle, preserving the integrity of the infrastructure contract between code and reality [7].

Infrastructure as Code brings to the operations layer the same pace of change as it does to the application layer; consequently, it is logical that its primary transport becomes a full-fledged CI/CD pipeline. A study of DevSecOps practices among 405 organizations revealed that 68% already perform automated infrastructure checks on every commit, and a further 12% plan to implement such checks within a year, as manual procedures are recognized as one of the main impediments to continuous delivery [13], as shown in Fig. 1.

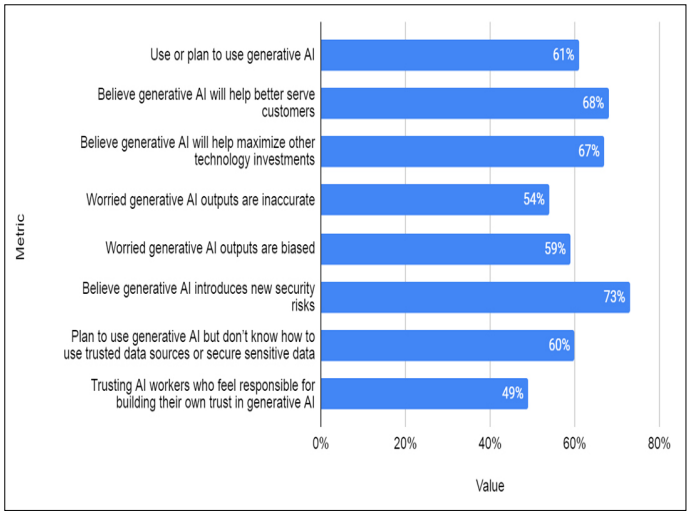

Fig. 1. Prevalence, Impediments, and Automation Gaps in Organizational DevSecOps Adoption [13]

The practical inference from these data is straightforward: the advantage of rapid and reliable infrastructure changes manifests only when Terraform is invoked automatically and triggered by the same events as application tests.

GitHub Actions provide a typical example. In the workflow, a single job is defined that first validates source code, then runs hashicorp/setup-terraform—locking the CLI version and caching providers—after which it executes terraform init, terraform plan, and, subject to review, terraform apply. To mitigate operational risk, permanent AWS keys are replaced by federation via OpenID Connect: GitHub issues a short-lived JWT, AWS STS exchanges it for temporary credentials, and permissions are granted via an IAM role bound to the repository. This approach removes static secrets from the repository, automatically limits the lifespan of tokens, and adheres to the principle of least privilege [14, 15].

Alternative platforms implement the same sequence using analogous commands. Azure Pipelines utilizes the Terraform Installer task, and authentication can occur either via a service principal or through OIDC, thereby eliminating the need for manual certificate management [16]. GitLab CI provides a built-in validate-plan-apply component that incorporates validation, planning, and apply stages, records the plan as an artifact, and displays it in the merge request interface; by default, state is stored in a protected object store, though any backend may be specified, including HCP Terraform or S3 [17], as illustrated in Fig. 2.

```
include:
  - component: gitlab.com/components/opentofu/validate-plan-apply@<VERSION>
    inputs:
      version: <VERSION>
      opentofu_version: <OPENTOFU_VERSION>
      root_dir: terraform/
      state_name: production

stages: [validate, build, deploy]
```

Fig. 2. Declarative CI/CD Pipeline Configuration for Terraform Workflows with OpenTofu Integration [17]

Bitbucket Pipelines follows the same steps, also leveraging OIDC federation for AWS access, which is configured via a CloudFormation template or the IAM console. Temporary credentials are injected into the container executing' terraform apply' and automatically expire upon completion of the step [18]. In every case, the pipeline remains concise: Terraform drives the logic, and the CI system orchestrates invocations and stores artifacts.

A key security element is the complete elimination of long-lived secrets. The OIDC flow has become an industry standard: GitHub, GitLab, and Bitbucket emit a signed token containing the repository and branch hashes, which the cloud provider maps to an IAM role. As a result, permissions are granted solely to the workflow performing validation and planning of the Terraform configuration, minimizing the risk of unauthorized access or secret leakage [14, 19]. The following example in Fig. 3 illustrates exchanging an OIDC ID token with Azure to obtain an access token for accessing cloud resources.

```
name: Run Azure Login with OIDC
on: [push]

permissions:
  id-token: write
  contents: read
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - name: 'Az CLI login'
        uses: azure/login@a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0
        with:
          client-id: ${{ secrets.AZURE_CLIENT_ID }}
          tenant-id: ${{ secrets.AZURE_TENANT_ID }}
          subscription-id: ${{ secrets.AZURE_SUBSCRIPTION_ID }}

      - name: 'Run az commands'
        run: |
          az account show
          az group list
```

Fig. 3. Exchanging an OIDC ID token with Azure to receive an access token [19]

Ultimately, the decision between HCP Terraform and a self-hosted backend hinges on the control versus operational cost trade-off. The managed service offers VCS integration, automatic plan distribution across workspaces, background drift detection, and health assessments. When the actual state deviates from the declared state, it opens a pull request with a corrective plan, thereby closing the GitOps loop. Self-hosted Terraform Enterprise delivers the same functionality but is deployed within the corporate network segment—critical under stringent regulatory or export controls; the user gains full control over versioning, hardware scaling, and Sentinel policies, but assumes responsibility for operating the Postgres database, Nomad queue, and object storage [20]. In both cases, the backend remains the single source of truth, and the CI pipeline retains only a reference to the workspace. Therefore, migrating between SaaS → self-hosted or vice versa requires no pipeline rewrites, affording organizations flexibility in risk management.

Practical infrastructure automation almost invariably begins with establishing the network perimeter, since proper segmentation underpins availability and enforcement of security policies. In Terraform, these tasks are addressed by a standard virtual private network module, which encapsulates the creation of subnets, route tables, NAT gateways, and traffic logs. The engineer specifies only logical parameters—names, tags, access tiers—and receives a reproducible topology that can be redeployed across regions and scaled without altering the overall code structure.

The next natural step is the deployment of a container orchestrator. Kubernetes has long been regarded as the standard for cloud-native workloads, and the Terraform ecosystem provides ready-to-use modules for various providers, allowing one to describe a cluster, node pools, and network policies in the same HCL language. This consolidation within a single repository eliminates version mismatches between the networking and compute layers. The cluster is provisioned by a single command alongside the underlying infrastructure, and access to image registries and load balancers is configured automatically via resource graph dependencies.

Immediately following the compute layer, observability components are added so that the platform does not remain silent. Monitoring and logging modules leverage the cloud's native services: system metrics are sent to a managed store, logs are collected by an agent and sent to streaming processors or a centralized repository. All configuration is defined in code, so when a new environment is instantiated, alerting parameters, collection intervals, and retention policies are inherited without additional effort. This ensures support teams receive a homogeneous signal path for incident investigation.

The final element of typical scenarios is the isolation of development, testing, and production lifecycles. The recommended practice is to associate a separate working

directory or module with a distinct Terraform workspace, thereby separating states and allowing independent release cadences. Developers may freely experiment within their workspace, automated checks run against the staging environment, and production-contour changes are permitted only from a protected branch after review. This maintains a single point of truth and avoids mixing settings across steps, which reduces the likelihood of errors and makes it faster to revert to a good state. If you sum up Terraform's main benefits—from how it describes what should be, having a single state file to an able system that offers and links well with CI/CD—it is clear this tool cuts down on risk in operations and quickens time-to-market. The capabilities afforded by provider-defined functions and modular architecture, together with built-in drift detection mechanisms, form a robust foundation for scalable, controlled automation. OIDC federation integration, along with state management via HCP Terraform or a self-hosted backend, provides high levels of security and flexibility when operating in hybrid and multi-cloud environments. Collectively, these features transform Terraform from a mere orchestration tool into a comprehensive Infrastructure Lifecycle Management platform, in which planning, validation, application, and change control are unified into a single, deterministic chain.

CONCLUSION

This study has proved that using Terraform as an IaC tool greatly improves the trustworthiness and foresight of managing cloud infrastructure. The described resource-state model and included dry-run planning mechanism make sure that any changes are deterministic and keep the risks of human errors at a minimum, shown by the coming in of provider-defined functions and a safe refactoring API. The presence of a single state file and remote backend solutions guarantees state integrity and consistency. At the same time, background drift analysis and health assessments enable the timely detection and correction of deviations from the declared configuration.

Terraform's ecosystem—comprising over 5,000 providers and a modular architecture—simplifies integration with diverse cloud and on-premises solutions. The ability to package patterns into reusable modules accelerates development and testing. Integration with CI/CD pipelines (GitHub Actions, Azure Pipelines, GitLab CI, Bitbucket Pipelines) and the use of an OIDC authentication flow eliminate long-lived secrets, adhering to the principle of least privilege, and markedly improve the security posture of the operational environment.

The platform's evolution toward Infrastructure Lifecycle Management—from discrete apply/destroy operations to a full cycle of planning, validation, application, and monitoring—positions Terraform as the foundation of a managed infrastructure-change pipeline. The Stacks concept in HCP Terraform which has Centralized Policy Enforcement, Automated Migrations, and Centralized Drift-Control Services creates a unified contract between development and operations. This allows the scaling of cloud workloads without increasing operational costs proportionally. Therefore, this research confirms that the adoption of Terraform will not only address the challenges of reproducibility and configuration control in complex multi- and hybrid-cloud environments but will also enable the foundation for continuous automated infrastructure management.

REFERENCES

1. V. Nedunoori, "The Cost of Cloud Misconfigurations: Preventing the Silent Threat," Information Week, Dec. 02, 2024. https://www.informationweek.com/it-infrastructure/the-cost-of-cloud-misconfigurations-preventing-the-silent-threat (accessed May 22, 2025).
2. J. Gregory, "2024 cloud threat landscape report: How does cloud security fail," IBM, Jan. 22, 2025. https://www.ibm.com/think/insights/2024-cloud-threat-landscape-report-how-does-cloud-security-fail (accessed May 23, 2025).
3. "Pulumi vs. Terraform vs. CloudFormation: Which IaC Tool is Best for Your Infrastructure?" Firefly, 2024. https://www.firefly.ai/academy/pulumi-vs-terraform-vs-cloudformation-which-iac-tool-is-best-for-your-infrastructure (accessed May 24, 2025).
4. M. Campbell, "Terraform 1.8 Adds Provider-Defined Functions, Improves AWS, GCP, and Kubernetes Providers," InfoQ, May 02, 2024. https://www.infoq.com/news/2024/05/terraform-provider-functions/ (accessed May 26, 2025).
5. "Providers," Terraform. https://registry.terraform.io/browse/providers (accessed May 26, 2025).
6. "Stacks overview," HashiCorp Developer, 2024. https://developer.hashicorp.com/terraform/cloud-docs/stacks (accessed May 27, 2025).
7. "Use health assessments to detect infrastructure drift," HashiCorp Developer, 2024. https://developer.hashicorp.com/terraform/tutorials/cloud/drift-detection (accessed May 28, 2025).
8. "HashiCorp Expands Unified Lifecycle Management for Hybrid Cloud," Globe Newswire News Room, 2025. https://www.globenewswire.com/news-release/2025/06/03/3092497/0/en/HashiCorp-Expands-Unified-Lifecycle-Management-for-Hybrid-Cloud-Operations.html (accessed May 29, 2025).
9. "Overview of the core," HashiCorp Developer, 2024. https://developer.hashicorp.com/terraform/intro/core-workflow (accessed May 30, 2025).
10. "Backend Type: s3," HashiCorp Developer, 2024. https://developer.hashicorp.com/terraform/language/backend/s3 (accessed Jun. 01, 2025).

11. "State: Sensitive Data," HashiCorp Developer, 2024. https://developer.hashicorp.com/terraform/language/state/sensitive-data (accessed Jun. 02, 2025).
12. "Manage Terraform configurations," HashiCorp Developer, 2024—https://developer.hashicorp.com/terraform/enterprise/workspaces/configurations (accessed Jun. 03, 2025).
13. J. Cheenepalli, J. D. Hastings, K. M. Ahmed, and C. Fenner, "Advancing DevSecOps in SMEs: Challenges and Best Practices for Secure CI/CD Pipelines," arXiv, 2025. https://arxiv.org/abs/2503.22612 (accessed Jun. 04, 2025).
14. "OIDC federation," AWS Identity and Access Management, 2025. https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_providers_oidc.html (accessed Jun. 05, 2025).
15. "Configuring OpenID Connect in Amazon Web Services," GitHub Docs, 2025. https://docs.github.com/en/actions/security-for-github-actions/security-hardening-your-deployments/configuring-openid-connect-in-amazon-web-services (accessed Jun. 06, 2025).
16. "Overview of Terraform on Azure," Microsoft, Nov. 11, 2024. https://learn.microsoft.com/en-us/azure/developer/terraform/overview (accessed Jun. 06, 2025).
17. "Infrastructure as Code with OpenTofu and GitLab," Gitlab. https://docs.gitlab.com/user/infrastructure/iac/ (accessed Jun. 06, 2025).
18. "Deploy on AWS using Bitbucket Pipelines OpenID Connect," Atlassian. https://support.atlassian.com/bitbucket-cloud/docs/deploy-on-aws-using-bitbucket-pipelines-openid-connect/ (accessed Jun. 07, 2025).
19. "Configuring OpenID Connect in Azure," GitHub Docs, 2025. https://docs.github.com/en/actions/security-for-github-actions/security-hardening-your-deployments/configuring-openid-connect-in-azure (accessed Jun. 09, 2025).
20. "Terraform Editions," HashiCorp Developer, 2024. https://developer.hashicorp.com/terraform/intro/terraform-editions (accessed Jun. 09, 2025).