https://doi.org/10.31891/csit-2026-2-3

Denys KOLOMYTSKYI
Master student majoring in Information Systems and Technologies, Khmelnytskyi National University
https://orcid.org/0009-0008-4335-1982
e-mail: kolomitskiy@gmail.com
Pavlo REHIDA
PhD in Computer Engineering, Associate Professor of the Department of Computer Engineering & Information Systems Department, Khmelnytskyi National University
https://orcid.org/0000-0002-6591-7069
e-mail: pavlo.rehida@gmail.com
Oksana ONYSHKO
Candidate of Pedagogical Sciences, Associate Professor of the Department of Software Engineering, Khmelnytskyi National University
https://orcid.org/0000-0002-2125-4160
e-mail: Van4o@ukr.net
Yuliia ILCHYSHYNA
Master student majoring in Information Systems and Technologies,
Khmelnytskyi National University
https://orcid.org/0009-0000-4372-0161
e-mail: juliaillchishina@gmail.com

Received: 07/04/2026
Accepted: 11/05/2026
Published: 31/05/2026

UDC 004.75

GENERALIZED METHOD FOR MANAGING THE LIFECYCLE OF TERRAFORM INFRASTRUCTURE ACROSS MULTIPLE ENVIRONMENTS

The article examines the challenge of managing the lifecycle of cloud infrastructure, as described using Terraform, across multiple environments (development, staging, and production). In industrial settings, multi-environment infrastructure management gives rise to a set of interrelated challenges: the absence of formalized procedures for promoting changes between environments with explicit source readiness verification, fragmented compliance checking that fails to distinguish code-level syntax validation from plan-level semantic verification, and configuration drift detection that lacks classification by severity, generating false alerts for expected changes such as dynamic IP addresses and rotated certificates. An analysis of existing approaches to environmental isolation, configuration compliance verification, and drift detection reveals that none of the current methods address the full lifecycle in a unified manner. A generalized method is proposed, based on a formalized two-projection lifecycle model: the horizontal projection describes an eight-stage finite automaton of an individual environment (Init, Author, Validate, Plan, Comply, Approve, Apply, Monitor), while the vertical projection defines a partially ordered environment space with a formalized promotion operation. The method introduces a source readiness precondition requiring the source environment to be in the monitoring stage with empty planning and drift deltas, versioned configuration snapshots ensuring code identity across environments, multi-level compliance verification across code, plan, and state levels using hierarchical policy inheritance, and a three-class drift classification (critical, actionable, informational) with an effective delta mechanism that filters expected changes. Experimental verification on a real multi-environment AWS infrastructure (27 managed resources, 3 environments) using the Scalr platform confirms that the proposed method ensures automated detection of 100% of policy violations before the apply stage (compared to 50% for GitOps CI/CD and 0% for Terraform CLI), reduces false drift alerts to zero, and decreases the number of manual promotion steps to a single approval for the production environment.

Keywords: Infrastructure as Code, Terraform, lifecycle management, multi-environment infrastructure, configuration drift, compliance verification, DevOps, cloud computing.

Overview of the Problem and Its Connection to Major Scientific and Practical Challenges

Modern approaches to software development have experienced a significant shift under the influence of the "Everything as Code" paradigm—a systematic method in which any artifact of a software system, including infrastructure, configuration, and security policies, is described as versioned code [1]. At the core of this paradigm is the practice of Infrastructure as Code (IaC), which involves declaratively describing cloud environment resources in configuration files stored in a version control system and executed by automated tools. Terraform is the leading tool in this category because of its provider-agnostic design and explicit state management through the terraform.tfstate file, which maps the described resources to the actual entities in the cloud [2].

A typical industrial infrastructure spans multiple environments—development, testing, and production—each consisting of isolated cloud resources at different stages of a software product's lifecycle. Managing this multi-environment setup introduces a complex set of interconnected issues that cannot be addressed by individual tools alone.

Current methods for environment isolation—based on workspaces, directory structures, or repository branches—balance

different trade-offs between isolation levels and maintenance complexity [3]. However, none of these methods offers a formal process for promoting changes across environments with clear prerequisites for source readiness. In practice, this can allow a change to reach the production environment from a source that may have undetected configuration drift or an incomplete application, raising the risk of failed deployments. An empirical study of versioning and rollback strategies for IaC templates confirmed that formalizing these procedures can significantly decrease the average recovery time after failures by an order of magnitude [4].

The verification of infrastructure configuration compliance remains scattered. Typical CI/CD pipelines include static security analysis during code validation but do not separate the checking of configuration syntax from verifying the semantics of planned changes as distinct stages. A configuration that is syntactically correct and has no known vulnerabilities can still break organizational policies only at deployment—such as using a forbidden instance type or planning to delete a protected resource. Additionally, studies have shown that adding declarative security policies directly into the resource provisioning pipeline can achieve a 92% improvement in compliance metrics [5].

Configuration drift—the gap between the actual state of cloud resources and what is defined in the code—is another key issue caused by manual changes in the cloud console, partial applications, or external processes outside of Terraform. Existing tools either lack a built-in way to monitor drift or identify discrepancies without prioritizing their severity, leading to a flood of false alerts about expected changes—such as dynamic IP addresses or automatically rotated certificates—which diminishes the response to truly critical issues.

Systematic reviews of IaC technologies show that, despite a considerable amount of research—from tool taxonomies to defect cataloging and formal analyses of idempotency—the comprehensive management of multi-environment infrastructure throughout its entire lifecycle remains neglected in academic studies [6]. There is no universal approach that covers the complete process from configuration setup to automatic drift correction, formalizes change propagation, and guarantees multi-level compliance verification at different stages of the pipeline. Creating such a method is an urgent scientific and practical challenge directly related to enhancing the reliability and security of cloud infrastructures in industrial settings.

Analysis of Recent Research and Publications

Pahl and his co-authors conducted the most thorough systematization of IaC technologies, proposing a classification based on four dimensions—application context, functionality, description language, and execution architecture—and developing a specialized DevOps lifecycle for infrastructure code based on it [7]. This cycle includes stages from code writing and validation to environment self-recovery; however, it is described at a conceptual level without formalizing transitions between stages and without considering coordination across multiple environments.

Based on a systematic review of the gray literature, Kumara and his co-authors identified the fundamental properties of high-quality infrastructure code: declarativeness, idempotence, and reproducibility [8]. Hanappi provided formal confirmation of these properties, demonstrating through state transition graphs that violations of idempotence are a hidden source of errors that only become apparent upon repeated application of configurations [9]. These findings serve as the theoretical foundation for formalizing the lifecycle but do not translate into a specific method for managing environments.

Empirical studies have uncovered systemic gaps in the IaC ecosystem, including fragmentation of tools, a lack of mature testing methods, and a shortage of standardized state management practices [10]. A qualitative analysis of testing practices identified six categories of checks—ranging from static analysis to policy compliance testing—yet none of the existing pipelines implement them as architecturally independent stages with distinct input artifacts [11].

Based on an analysis of 1,448 commits, Rahman and his co-authors developed a taxonomy of eight categories of infrastructure code defects, with configuration data errors being the most common and violations of idempotency being the most specific to IaC [12]. Another area of focus is the detection of configuration drift: Dinu and Fontaine proposed a method for continuously comparing Terraform plan results with an expected empty plan and automatically notifying of discrepancies, although without classifying the detected drift by severity level [13].

In the realm of state management, the fork of the Terraform ecosystem following HashiCorp's license model change is important: the open-source fork OpenTofu, supported by the Linux Foundation, maintains compatibility at the HCL language and provider levels [14], which requires provider independence in any general method.

Research on GitOps pipelines for Terraform shows that deployment frequency increases and recovery time decreases compared to traditional methods; however, the pull synchronization model is limited to planning and application stages without formalizing progress between environments [15].

Thus, existing studies focus on specific aspects—tool classification, formalization of idempotency, testing, drift detection, and GitOps automation—but do not provide a comprehensive method that covers the entire lifecycle of a multi-environment Terraform infrastructure, including formalized change propagation, multi-level compliance verification, and classification of configuration drift.

Statement of the article's objectives

The purpose of this article is to create a comprehensive approach for managing the lifecycle of Terraform infrastructure across multiple environments. To achieve this, three tasks are outlined: (1) formalize a two-projection lifecycle model as a finite state machine and a partially ordered environment space, covering stages from configuration setup to detecting and fixing configuration drift; (2) justify a method for propagating changes between environments with source readiness requirements, multi-level compliance checks, and a versioned rollback system; (3) test the approach on a real multi-environment infrastructure and evaluate its effectiveness using specific metrics.

Conceptual Model and Formalization of the Method

Formalization of Concepts and the Lifecycle Model

Let M be a finite set of Terraform modules, where each module $m = \langle R_m, V_m, O_m \rangle$ is defined by a set of resource descriptions R_m, input variables V_m and output values O_m. An infrastructure configuration is defined as a pair:

$$C = \langle m_{root}, \Theta \rangle, \quad (1)$$

where $m_{root} \in M$ — is the root module (entry point), and, $\Theta = \{\theta_1, \dots, \theta_k\}$ — is the set of input variable values that parameterizes a specific instance of the infrastructure. It is Θ that determines the differences between environments when the root module is shared.

The environment is defined by four factors:

$$e = \langle id_e, C_e, S_e, P_e \rangle, \quad (2)$$

where id_e — is a unique identifier; C_e — is the configuration; S_e — is the current state; P_e — is the set of compliance policies. The inclusion of P_e in the definition reflects the principle of built-in compliance: policies are an integral part of the environment.

The concept of state involves distinguishing three entities. The desired state $S_e^d = eval(C_e)$ is set by interpreting the configuration. The recorded state S_e^r— is the content of terraform.tfstate, which updates after each successful operation. The actual state S_e^a— includes resources that presently exist in the cloud and may differ from S_e^r due to changes outside of Terraform. This three-part model allows us to formally define the *planning delta* $\Delta_e^{plan} = S_e^d \setminus S_e^r$ and the drift delta as a symmetric difference:

$$\Delta_e^{drift} = S_e^r \triangle S_e^a, \quad (3)$$

which covers both resources that are modified or deleted outside of Terraform and resources added to the cloud without a matching description in the configuration. The environment is *consistent* if both deltas are empty.

A set of environments forms the environment space $E = \{e_1, \dots, e_n\}$ with a partial order relation $\preceq$. For a typical configuration such as $e_{dev} \preceq e_{staging} \preceq e_{prod}$ a change reaches the production environment only after passing through all intermediate environments.

The lifecycle of a single environment is modeled as a finite state machine $A_e = \langle \Sigma, \delta, \sigma_1, F \rangle$ with eight stages: σ_1 (Init) — initializing the backend and providers; σ_2 (Author) — making configuration changes; σ_3 (Validate) — performing syntax checks and static security analysis; σ_4 (Plan) — calculating Δ_e^{plan} without applying it; σ_5 (Comply) — verifying the plan's semantics for policy compliance; σ_6 (Approve) — obtaining approval (manual for production, automatic for development); σ_7 (Apply) — implementing the plan, which updates $S_e^r \leftarrow S_e^a$; σ_8 (Monitor) — periodically calculating Δ_e^{drift}. A failure at stages σ_3, σ_5 or σ_6 causes the automaton to return to σ_2 to correct the configuration, while detecting drift at σ_8 triggers a transition back to σ_4 , completing the feedback loop. The automaton's structure and feedback transitions are illustrated in Fig. 1.

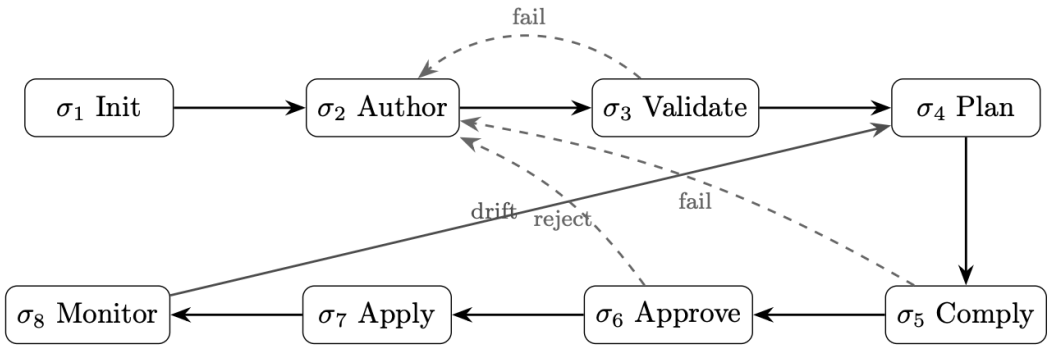

Fig. 1. Finite-state machine A_e depicts the life cycle of a single environment. Solid arrows show the primary transition sequence; dashed arrows indicate backtransitions in case of failure; the transition $\sigma_8 \rightarrow \sigma_4$ – illustrates the drift elimination cycle.

A key difference from existing conveyors is the architectural separation of σ_3 and σ_5: the former verifies the predicate $valid(C_e, P_e^{code})$ without consulting the provider, while the latter verifies$comply\left(\Delta_e^{plan}, P_e^{plan}\right)$ on the plan artifact with computed resource attributes.

Promotion and Rollback Mechanism

The promotion operation between adjacent environments $e_i \preceq e_j$ is defined as:

$$promote(e_i, e_j): C_{e_j} \leftarrow \left\langle m_{root}, \Theta_{e_j} \right\rangle, \tag{4}$$

where the root module remains shared, and Θ_{e_j} reflects the specifics of the target environment:

$$ready(e_i) \Leftrightarrow stage(e_i) = \sigma_8 \wedge \Delta_{e_i}^{plan} = \emptyset \wedge \Delta_{e_i}^{drift} = \emptyset, \tag{5}$$

in other words, the source environment is in the monitoring phase with empty deltas. None of the existing approaches automatically verify this condition.

Code identity between environments is ensured through *versioned configuration snapshots* $\widehat{C_e^v} = \langle m_{root}, \Theta_e, v \rangle$ where v — is the version number assigned after successfully completing a full cycle in the source environment. A snapshot is an immutable artifact that prevents implicit changes in the dependencies between testing in e_i and deployment in e_j.

After advancing to the target environment, stages $\sigma_3 \rightarrow \sigma_5 \rightarrow \sigma_6 \rightarrow \sigma_7$ are executed sequentially with policies P_{e_j}, ensuring independent compliance verification for each environment.

The rollback mechanism relies on the history of state snapshots $H_e = \{S_e^{r,1}, \ldots, S_e^{r,v}\}$. The $rollback(e, v')$ operation restores the saved state$S_e^{r,v'}$ followed by the cycle $\sigma_4 \rightarrow \sigma_7$ to bring the actual state into compliance. The safety of the rollback is determined by the predicate:

$$safe(e, v, v') \Leftrightarrow \forall r \in \left(S_e^{r,v} \setminus S_e^{r,v'}\right): rev(r) = 1, \tag{6}$$

where $rev(r) = 1$ for reversible resources (computational instances, network rules) and$rev(r) = 0$ for irreversible ones (databases with accumulated transactions). If $safe = false$ the σ_6 approval stage is forcibly switched to manual mode.

Multi-level compliance checking and drift classification

The set of environment policies is divided according to compliance levels:

$$P_e = P_e^{code} \cup P_e^{plan} \cup P_e^{state}, \tag{7}$$

where P_e^{code} applies to σ_3 (syntax, hardcoded secrets, open ports), P_e^{plan} – to σ_5 (prohibited instance types, deletion of protected resources, cost estimation), and P_e^{state} — to σ_8 (compliance of the actual state with security requirements). Policies are organized into an inheritance hierarchy: $P_e = P_{global} \cup P_{env(e)} \cup P_{comp(e)}$, with strictness increasing along $\preceq$ — the same policy may be advisory in e_{dev} and hard-mandatory in e_{prod}.

Detected drift is categorized into three types:

$$class\left(r, \Delta_e^{drift}, P_e^{state}\right) \in \{\text{critical, actionable, informational}\}, \tag{8}$$

Critical drift (a violation of a strict mandatory policy) is automatically resolved by $\sigma_8 \rightarrow \sigma_4$. Actionable drift (a discrepancy that does not breach critical policies) requires manual confirmation. Informational drift (expected changes such as dynamic IP addresses or rotated certificates) is excluded from the effective delta through the set of ignored attributes I_e:

$$\Delta_e^{eff} = \Delta_e^{drift} \setminus \{r: attr(r) \in I_e\}, \tag{9}$$

which reduces false alerts without compromising oversight of critical changes.

Implementation and Experimental Evaluation of the Method

To validate the method, we selected platform [16]—an industrial-grade Terraform/OpenTofu infrastructure orchestration system that features a directory-based isolation pattern, hierarchical configuration inheritance, and built-in compliance checking via Open Policy Agent (OPA). The experimental infrastructure is deployed in Amazon Web Services (AWS) cloud and includes a typical web service: network layer (VPC, subnets, security groups), compute layer (EC2), storage layer (RDS PostgreSQL, S3), and access management (IAM). The configuration is organized into a library of five submodules, totaling approximately 1,200 lines of HCL and 27 managed resources. Environment space: $e_{dev} \preceq e_{staging} \preceq e_{prod}$.

The platform's Run pipeline directly maps to the A_e: state machine: two independent OPA policy checkpoints correspond to stages σ_3 (configuration check before planning) and σ_5 (check of the JSON plan artifact after planning). Monitoring of σ_8 is carried out through drift detection—periodic execution of `terraform plan` in refresh-only mode. Manual execution of the Terraform CLI and a typical GitOps CI/CD pipeline using GitLab CI, integrated with a static security analyzer, were chosen as comparison methods.

Experiment 1: Multi-level compliance verification.

Four violation scenarios were identified: two code-level violations—(C1) a hardcoded AWS secret and (C2) a security group with a 0.0.0.0/0 rule on port 22; and two plan-level violations—(C3) a prohibited m5.24xlarge instance type and (C4) deletion of a protected RDS instance. Each violation was added as a separate commit to the $e_{staging}$ configuration.

Table 1

The stage of detecting violations by approach

Scenario	Level	Terraform CLI	GitOps CI/CD	Method (Scalr)
C1	P^{code}	Not automated	σ_3	σ_3(OPA)
C2	P^{code}	Not automated	σ_3	σ_3(OPA)
C3	P^{plan}	Not automated	Not automated	σ_5(OPA)
C4	P^{plan}	Not automated	Not automated	σ_5(OPA)

The proposed method achieves $R_{pre-apply} = 1{,}0$ – all four violations are detected automatically before the apply stage. GitOps CI/CD detects only code-level violations ($R_{pre-apply} = 0{,}5$), because the static analyzer does not process the plan artifact. The Terraform CLI has no built-in mechanisms for automated policy verification ($R_{pre-apply} = 0{,}0$): violations may be noticed by the operator during a manual review of the terraform plan output, but this depends on attentiveness and experience, and it is not a systematic check. The method's detection time is 15–40 seconds for code-level violations and 45–90 seconds for plan-level violations.

Experiment 2: drift detection and classification.

Three types of external changes were simulated via the AWS console: (D1) critical – adding the rule 0.0.0.0/0:443 to the prod-environment security group; (D2) operational – changing the EC2 instance tag; (D3) informational – changing the private IP address after a restart. The drift detection interval was set to $\tau = 5$ minutes to accelerate data collection (the typical value for production environments is 1–24 hours, depending on organizational requirements).

Table 2

Drift Detection and Classification Results

Scenario	Class	Terraform CLI	GitOps CI/CD	Метод (Scalr)
D1	Critical	Not automated	Not automated	Auto-resolution, 6.2 min.
D2	Operational	Not automated	Not automated	Notification, awaiting confirmation
D3	Informational	Not automated	Not automated	Excluded from Δ_e^{eff}

The Terraform CLI and GitOps CI/CD lack built-in automated drift monitoring; detection requires manually running `terraform plan` or configuring a separate cron job. The proposed method accurately identified all three types of drift. Critical drift (D1) was automatically addressed in 6.2 minutes, including the detection cycle wait and the full $\sigma_4 \to \sigma_7$. Informational drift (D3) was filtered out from the effective delta via the I_e set, ensuring $F_{false} = 0$ – no false positives.

Experiment 3: Automation of the promotion process.

For the full promotion cycle $e_{dev} \preceq e_{staging} \preceq e_{prod}$ we recorded the number of manual steps N_{manual} (each CLI command or operator action requiring manual input counts as one step) and whether there was an automated check for the readiness of the source environment.

Table 3

Comparison of promotion automation

Characteristics	Terraform CLI	GitOps CI/CD	Method (Scalr)
N_{manual}	9	3	1
Validation of $ready(e_i)$	None	None	Automatic
Policy validation during deployment	None	Partial (P^{code})	Full (P^{code}, P^{plan})

For Terraform CLI, $N_{manual} = 9$: the operator executes a plan + apply sequence for each of the three environments (6 commands) and reviews the plan output before each apply (3 reviews). For GitOps CI/CD $N_{manual} = 3$: manually launching the pipeline or merging for each environment without automatic readiness verification $ready(e_i)$. The proposed method requires $N_{manual} = 1$ – approval of the application only for the production environment; the remaining stages, including source readiness verification and cascading propagation, are automated.

Conclusions from This Research and Prospects for Further Study

This article introduces a generalized approach for managing the lifecycle of Terraform infrastructure across multiple environments. It relies on a formalized two-projection model: the horizontal projection outlines an eight-state finite automaton for a single environment, while the vertical projection represents a partially ordered space of environments with a formal transition operation.

A key theoretical contribution is the architectural separation of the verification of code compliance σ_3 and plan semantics σ_5 into independent stages with different input artifacts—something none of the approaches considered achieve. The formalization of the source readiness predicate $ready(e_i)$ or versioned configuration snapshots and the rollback safety predicate$safe(e, v, v')$ ensures atomicity of progression and controlled rollback, taking into account resource reversibility. A three-class drift classification with an effective delta enables differentiated responses: automatic remediation of critical violations, confirmation for operational violations, and filtering of informational violations.

Experimental verification on a real AWS infrastructure (27 resources, 3 environments) confirmed that the method provides automated detection of 100% of policy violations prior to the deployment stage (compared to 50% in GitOps CI/CD and 0% in Terraform CLI), reduces false drift alerts to zero, and reduces manual deployment steps to a single approval for the production environment.

The method has certain limitations. First, it depends on a directory-based isolation pattern and HCL semantics, which guarantees compatibility with Terraform and OpenTofu but excludes tools with fundamentally different architectures (stateless or agent-based). Second, experimental validation was conducted on a single platform (Scalr) that natively implements the required mechanisms; applying it to other platforms necessitates adapting the OPA integration and the drift detection mechanism. Third, the method does not include automated cost estimation of infrastructure changes as a formalized stage.

Prospects for future research include incorporating cost estimation as part of the automaton, adapting the method to nonlinear topologies of environment spaces with parallel propagation chains, and testing on alternative orchestration platforms.

ADDITIONAL INFORMATION

DECLARATION ON THE USE OF GENERATIVE ARTIFICIAL INTELLIGENCE TOOLS

During the preparation of this work, the authors used Grammarly in order to: grammar and spelling check; DeepL Translate in order to: some phrases translation into English. After using these tools and services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

AUTHOR CONTRIBUTIONS

Conceptualization, Denys Kolomytskyi, Pavlo Rehida and Oksana Onyshko; methodology, Denys Kolomytskyi and Pavlo Rehida; software, Denys Kolomytskyi and Oksana Onyshko; validation, Pavlo Rehida and Andriy Drozd; formal analysis, Andriy Drozd; investigation, Pavlo Rehida and Andriy Drozd; resources, Oksana Onyshko; data curation, Denys Kolomytskyi and Oksana Onyshko; writing – original draft preparation, Denys Kolomytskyi and Oksana Onyshko; writing – review and editing, Pavlo Rehida and Andriy Drozd.

1. Wei H., Madhavji N., Steinbacher J. *Understanding Everything as Code: A Taxonomy and Conceptual Model*. 2025. 12 p. (Preprint. arXiv; 2507.05100). DOI: 10.48550/arXiv.2507.05100.

2. *Terraform Documentation* / HashiCorp. 2026. Available at: https://developer.hashicorp.com/terraform/docs (accessed: March 3, 2026).

3. Brikman Y. *How to manage multiple environments with Terraform*. Gruntwork Blog. 2022. Available at: https://medium.com/gruntwork/how-to-manage-multiple-environments-with-terraform-32c7bc5d692 (accessed: March 5, 2026).

4. Karanam R. *Multi-cloud IaC template versioning and rollback strategies: An empirical study with Terraform and GitOps*. World Journal of Advanced Research and Reviews. 2024. Vol. 22, no. 02. P. 2354–2363. DOI: 10.30574/wjarr.2024.22.2.1357.

5. Sharma A., Rodriguez E., Tanaka K., Johnson M. *Scaling Infrastructure Governance: A Policy-Driven Framework Combining Terraform Automation and Red Hat Satellite*. ResearchGate. 2026. Available at: https://www.researchgate.net/publication/399734521_Scaling_Infrastructure_Governance_A_Policy-Driven_Framework_Combining_Terraform_Automation_and_Red_Hat_Satellite (accessed: March 2, 2026).

6. Pahl C., Gunduz N. G., Sezen O. C., Ghamgosar A., El Ioini N. *Infrastructure as Code – Technology Review and Research Challenges*. Proceedings of the 20th International Conference on Cloud Computing and Services Science (CLOSER 2025). 2025. P. 1–8. DOI: 10.5220/0012625200003851.

7. Pahl C., Gunduz N. G., Sezen Ö. C., Ghamgosar A., Hofer F., El Ioini N. *A Systematic Review of Infrastructure-as-Code Technologies*. Proceedings of the 15th International Conference on Cloud Computing and Services Science (CLOSER 2025). 2025. Vol. 1. P. 151–158.

8. Kumara I., Garriga M., Romeu A. U., Di Nucci D., Palomba F., Tamburri D. A., van den Heuvel W. J. *The Do's and Don'ts of Infrastructure Code: A Systematic Gray Literature Review*. Information and Software Technology. 2021. Vol. 137. Art. 106593. DOI: 10.1016/j.infsof.2021.106593.

9. Hanappi O., Hummer W., Dustdar S. *Asserting Reliable Convergence for Configuration Management Scripts*. Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016). 2016. P. 328–343. DOI: 10.1145/2983990.2984000.

10. Guerriero M., Garriga M., Tamburri D. A., Palomba F. *Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry*. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). Cleveland, OH, USA, 2019. P. 580–589. DOI: 10.1109/ICSME.2019.00092.

11. Hasan Mohammed Mehedi, Bhuiyan Farzana Ahamed, Rahman Akond. *Testing Practices for Infrastructure as Code*. LANGETI 2020: Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing. 2020. P. 7–12. DOI: 10.1145/3416504.3424334.

12. Rahman A., Farhana E., Parnin C., Williams L. *Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts*. Proceedings of the 42nd International Conference on Software Engineering (ICSE). 2020. P. 752–764. DOI: 10.1145/3377811.3380409.

13. Dinu F., Fontaine J.-M. *Infrastructure Drift Detection — How to Fix It with IaC Tools*. Spacelift Blog. 2024. Available at: https://spacelift.io/blog/drift-detection (accessed: March 3, 2026).

14. *OpenTofu Documentation* / OpenTofu. 2026. Available at: https://opentofu.org/docs/ (accessed: March 3, 2026).

15. Hughes L., Webb J., Morgan H., Song M. *GitOps for Continuous Deployment in Cloud-Native Infrastructure*. 2024. 5 p. Available at: https://www.researchgate.net/publication/392163663_GitOps_for_Continuous_Deployment_in_Cloud-Native_Infrastructure (accessed: March 3, 2026).

16. *Terraform Module Registry — Hierarchical Inheritance*. Scalr Blog. 2021. Available at: https://scalr.com/blog/hierarchical-terraform-module-registry (accessed: March 3, 2026).

Денис КОЛОМИЦЬКИЙ, Павло РЕГІДА, Оксана ОНИШКО, Юлія ІЛЬЧИШИНА
Хмельницький національний університет

УЗАГАЛЬНЕНИЙ МЕТОД КЕРУВАННЯ ЖИТТЄВИМ ЦИКЛОМ TERRAFORM-ІНФРАСТРУКТУРИ ДЛЯ КІЛЬКОХ СЕРЕДОВИЩ

У статті досліджується проблема управління життєвим циклом хмарної інфраструктури, описаної засобами Terraform, для кількох середовищ (розробки, тестування, виробництва). Здійснено аналіз існуючих підходів до ізоляції середовищ, перевірки відповідності конфігурацій та виявлення конфігураційного дрейфу. Запропоновано узагальнений метод, що ґрунтується на формалізованій двопроекційній моделі життєвого циклу у вигляді восьмистадійного скінченного автомата та частково впорядкованого простору середовищ. Метод включає формалізовану процедуру просування змін між середовищами з передумовою готовності джерела, багаторівневу перевірку відповідності на рівнях коду, плану та стану, а також трикласову класифікацію конфігураційного дрейфу з механізмом версіонованого відкату. Експериментальна верифікація на реальній мультисередовищній інфраструктурі AWS із використанням платформи Scalr підтверджує, що запропонований метод забезпечує виявлення 100% порушень політик до стадії застосування та зведення хибних сповіщень дрейфу до нуля через ефективну дельту.

Ключові слова: Infrastructure as Code, Terraform, управління життєвим циклом, мультисередовищна інфраструктура, конфігураційний дрейф, перевірка відповідності, DevOps, хмарні обчислення.