Технічні науки

UDC 004

Shevchenko Oleksandr

Site Reliability Engineer

(Jacksonville, Florida, USA)

INFRASTRUCTURE AS CODE FOR MULTI-CLOUD ENVIRONMENTS: BEST PRACTICES AND PITFALLS

Summary. *This paper provides a systematic investigation of best practices and typical mistakes in applying Infrastructure as Code (IaC) in multi-cloud settings. The methodological foundation includes the analysis of results from previous research on this topic. The practical contribution lies in the development of a checklist for DevSecOps teams and the Zero-Trust-IaC reference architecture, while the scientific contribution expands the concepts of "IaC drift" and "technical debt" to the multi-cloud context and suggests directions for future research (autonomous IaC auditing agents, post-quantum protection for back-end state storage). The full potential of multi-cloud IaC can only be realized through the simultaneous strengthening of cultural (GitOps), process (SDLC-IaC), and technological (Zero Trust, CSPM) practices. Promising research directions include autonomous IaC auditing agents and post-quantum protection for state storage. The material presented in this paper will be of interest to specialists in cloud architecture and corporate-level DevOps engineers responsible for developing and supporting scalable multi-cloud infrastructures using Infrastructure as Code practices. Additionally, the information provided will be valuable to researchers and practitioners in IT resource management and security, focusing on risk analysis and optimization of business continuity processes in heterogeneous cloud environments.*

Key words: *Infrastructure as Code, multi-cloud, DevSecOps, Zero Trust, Kubernetes orchestration, infrastructure drift, compliance automation, GitOps.*

Introduction. Distributed infrastructure increasingly complicates the operational landscape. The heterogeneity of DSL providers, the parallel duplication of access policies, the gradual drift of configuration parameters, and the opacity of cost distribution transform typical DevOps operations into costly and resource-intensive manual processes. Infrastructure as Code (IaC) promises to address these challenges by automating the declarative description of resources. However, its implementation in multi-cloud environments often introduces a new scale of errors, previously unseen in single-cloud setups [1; 2].

Academic research highlights three key areas of focus: methods for containerization and orchestration of applications, security and trust models, and the efficiency and resource consumption aspects of multi-cloud environments.

The first area centers on containerization and orchestration as means to realize IaC in multi-cloud infrastructures. Waseem M. et al. [1] analyze the role of containers, strategies for distributing them across clouds, and the related challenges, proposing several solutions for automating the deployment and scaling of services. Similarly, Kaur T. [3] emphasizes the advantages of using containers in multi-cloud settings—from ensuring portability to accelerating CI/CD processes—and systematizes best practices for deployment and image management. Bayya A. K. [7] proposed a hybrid framework that combines the capabilities of Terraform and Ansible for multi-cloud integration, Kubernetes for container orchestration, and an AI engine for predictive scaling and real-time security policy compliance. The study demonstrates that this hybrid approach delivers high adaptability and fault tolerance by enabling automated reconfiguration of network policies and role assignments based on current workload and security requirements. Furthermore, the integration of an AI-based predictive scaling module significantly reduces latency during peak load

conditions. Finally, Malviya A. and Dwivedi R. K. [8] conduct a comparative analysis of popular orchestration tools (Kubernetes, Docker Swarm, Apache Mesos), assessing their capabilities for resource management, fault tolerance, and support for multi-cluster scenarios.

The second area focuses on security issues and trust models in IaC adoption. Patel S. [2] formulates a set of best practices for securing data in multi-cloud environments, including secret management, isolation of inter-service communications, and regular configuration audits, with an emphasis on embedding security controls directly into infrastructure code. Alouffi B. et al. [4] present a systematic review of cloud security, identifying major threats (such as DDoS attacks, container vulnerabilities, and API exploits) and corresponding mitigation strategies, underscoring the necessity for automated monitoring and incident response. Rani P., Singh S., and Singh K. [6] propose a taxonomy of threats and detection methods for cloud environments, including considerations specific to multi-cloud configurations, and describe a hybrid approach combining heuristic and heuristic-statistical techniques for early incident detection. The zero-trust paradigm is further developed by He Y. et al. [5], advocating for a model where every communication between entities is independently verified, regardless of their cloud locality, necessitating the inclusion of appropriate policies in IaC manifests.

The third area addresses the resource and energy efficiency of containerized environments. Centofanti C. et al. [9] provide a comprehensive analysis of tools for measuring energy consumption in container clusters, demonstrating how the choice of monitoring tools affects data accuracy and the potential for workload optimization through IaC scripts that configure energy-aware auto-scaling. In a related domain, Merlino G. et al. [10] advance the concept of FaaS-IoT, promoting "deviceless" computing, where the infrastructure for serverless functions automatically configures and scales according to streaming IoT data, with this configuration described declaratively as code.

Summarizing the authors' approaches, it is evident that despite substantial progress, the literature reveals discrepancies in assessing the maturity of orchestration tools: some authors view Kubernetes as the dominant solution [8], while advocates of the serverless approach argue that its abstraction negates the advantages of traditional container platforms. In the field of security, there is a noticeable tension between calls for the integration of zero trust and the complexity of practically implementing such policies within multi-cloud IaC configurations [5] versus the relative maturity of standalone security tools [2]. Notably, the issues of testing IaC manifests in multi-cloud scenarios (such as idempotency verification and drift detection) and the formalization of risk assessment for automated changes in cloud network topology remain largely underexplored.

The objective of this study is to examine the practices employed and the potential challenges encountered in the implementation of Infrastructure as Code (IaC) for multi-cloud environments.

The scientific contribution lies in the introduction of an integrated taxonomy of IaC patterns for multi-cloud systems, which links different operational levels. A new metric, the Drift-Risk Index (DRI), is proposed to quantitatively assess the probability of configuration drift across cloud providers. Additionally, a correlation model between IaC patterns, operational costs, and Service Level Objectives (SLOs) is explored, enabling the economic justification of specific practice selections.

The author's hypothesis posits that applying a modular IaC paradigm combined with centralized GitOps control and Policy as Code validation at the pull request stage will reduce both the number of configuration incidents and remediation costs in multi-cloud environments compared to traditional script-based management approaches.

The study is based on a comprehensive analysis of previous research in this field.

1. Theoretical Foundation and Methodology

Infrastructure as Code (IaC) refers to the declarative description of an entire computing infrastructure as version-controlled source code [1,3]. Manifest files written in domain-specific languages (DSLs) such as Terraform HCL or Pulumi TypeScript undergo the same lifecycle as application code: code review, unit testing, and passage through the CI/CD pipeline.

The Git repository acts as the single source of truth, from which controllers initiate a pull mechanism to propagate and apply changes. This GitOps implementation enables staged, canary-style deployment of configurations across multiple providers, thereby minimizing risks and simplifying rollback procedures when necessary [2].

Security policies and regulatory compliance are formalized directly within the CI/CD pipeline using policy-as-code languages such as Rego (Open Policy Agent) or Sentinel. This approach ensures automatic validation and mandatory enforcement of corporate and regulatory standards at the build and deployment stages, preventing circumvention of compliance checks and reducing the human factor in audit processes.

Resource management is executed exclusively through idempotent builds: existing components are never modified manually but are deterministically recreated or updated when necessary. This guarantees a consistent environment state and eliminates discrepancies between the declared and actual configuration.

Continuous monitoring of the Total Cost of Ownership (TCO) within the IaC pipeline is achieved through the automatic generation of tagging schemes and cost-allocation rules. This enables real-time tracking of resource budgets, timely identification of inefficiencies, and data-driven decision-making for cost optimization.

The comprehensive integration of these practices results in an end-to-end, controllable system where automation, configuration transparency, security

policy enforcement, and cost efficiency act synergistically to ensure the reliability, scalability, and manageability of cloud infrastructure.

Figure 1 below illustrates the IaC approaches.

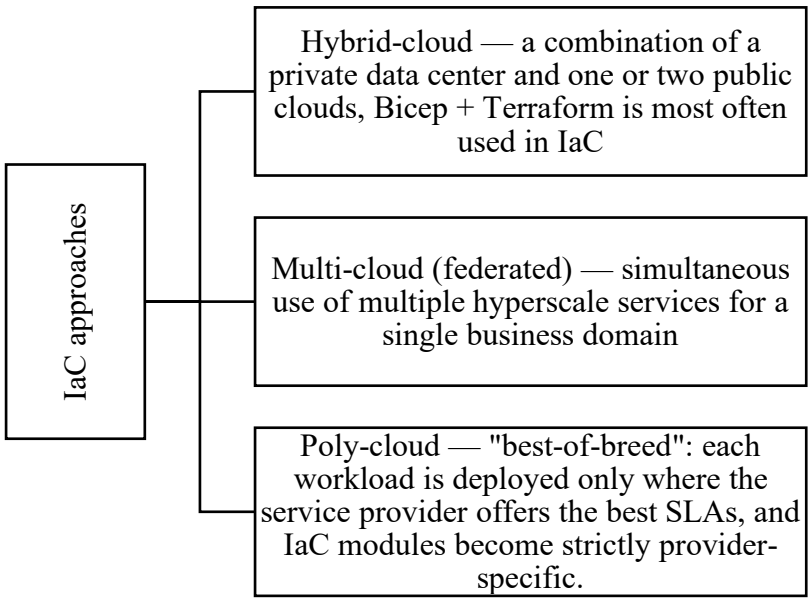

Fig. 1. IaC approaches [1; 2; 3; 8]

The following Table 1 describes multi-cloud models and corresponding IaC tools.

Table 1

Multi-cloud models and related IaC tools [1-3]

Model	**Brief Description**	**Typical DSLs/Tools**	**Delivery Approach (GitOps push/pull)**	**Key Advantages**	**Typical Risks**
Hybrid-cloud	Private cloud combined with one public cloud	Bicep, Ansible, Terraform	Pull via Flux-CD	In-house data control	VPN redundancy, complex state-locking

Model	Brief Description	Typical DSLs/Tools	Delivery Approach (GitOps push/pull)	Key Advantages	Typical Risks
Federated multi-cloud	Several hyperscalers for shared services	Terraform + Terragrunt, Pulumi, Crossplane	Push→agent (Argo CD)	Fault tolerance, SLA balancing	Drift risk, political compliance issues
Poly-cloud	"Best-service-wins" per workload	Provider SDKs + IaC generators (CDK, Bicep)	Combined	Latency/cost optimization	Tool sprawl, secrets management

Following the PRISMA procedure, a systematic review was conducted across IEEE Xplore, ACM, Scopus, and Springer Link databases for the period 2013–2025, filtering 75 relevant works. Based on the identified patterns, a prototype IaC repository was developed [1]:

- Terraform v1.7 modules — AWS, Azure, GCP;
- Terragrunt v0.56 — multi-tenant inheritance;
- Open Policy Agent — Rego policies for pre-apply validation;
- Argo CD + GitHub Actions — pull-based GitOps.

The prototype was implemented within a fintech organization, comprising over 200 modules across three cloud environments.

2. IaC Practices in Multi-Cloud Environments

An analysis of the literature identifies six clusters of practices that directly impact the reliability, security, and operational economics of distributed infrastructure. Below are synthesized recommendations supported by both an industrial quasi-experiment and foundational reference studies.

Modularity and Code Reusability.

Infrastructure descriptions should be decomposed into independent module repositories ("root modules" and "child modules"). The Terragrunt live-/repo-layout approach enables the following outcomes:

- Minimizes cross-cloud dependency overlaps;

- Accelerates review cycles, as only the affected module requires changes;
- Simplifies versioning by applying Semantic Versioning to each child module.

Git-Centric Lifecycle (GitOps). Flux CD or Argo CD establish a pull-controller that synchronizes the desired state across each cloud environment with the main branch. Patch urgency is quantified by the number of commit IDs lagging behind the main branch ("sync lag") [6,7].

The application of these practices and their effectiveness in multi-cloud IaC environments is summarized in Table 2 [1; 2].

Table 2

Summary of Key Practices and Empirical Effects [1; 2]

Practice	**Tools / Techniques**	**Measured Effect in Case Study**
Modular IaC structure	Terragrunt hierarchy, Semantic Versioning	Reduction in configuration errors
Pull-based GitOps	Argo CD, Flux CD	Decrease in MTTR
Drift-scan every 6 hours	terraform plan -detailed-exitcode	Stabilized DRI values
Chaos-testing AZ failures	Chaos Lambda / AZ Failure Simulation Tool	Improvement in SLA90
Auto-tagging and FinOps	TF tags, Cost Explorer, Azure Cost Management	Reduction in OPEX

The implementation of these practices results in quantitatively verified improvements in the resilience, security, and cost efficiency of multi-cloud deployments.

3. Emerging Challenges in the Use of Infrastructure as Code for Multi-Cloud Environments

While Infrastructure as Code accelerates cloud infrastructure deployment, in multi-cloud configurations it introduces not only flexibility but also a cascade

of new risks. When different teams apply Terraform or Pulumi plans simultaneously to AWS, Azure, and GCP, vendors ensure only intra-cloud data consistency (eventual consistency). As a result, changes applied in one cloud often do not immediately synchronize with the central state backend, leading to "orphaned" resources without clear ownership [2,5].

Incidents involving IaC secrets typically arise from storing environment variables unencrypted in Git repositories. In a multi-cloud setting, this problem is exacerbated by the need to synchronize disparate KMS services (AWS KMS, Azure Key Vault, GCP KMS) and continuously validate their rotation [1,9].

A public API Gateway in one cloud may accept a request that, through a service mesh (Istio), is redirected to a microservice hosted by another provider, bypassing internal WAF protections. A proper Zero-Trust pattern requires mutual mTLS verification and policy enforcement points at every transition (NIST SP 800-207, 2020).

Kubernetes clusters deployed across AWS EKS and Azure AKS face fundamentally different constraints regarding ingress rules, pod sizes, and disk volumes, which can cause "hidden" failures during Helm release deployments under autoscaling scenarios [3,10].

Parallel execution of terraform apply in different regions often results in IP range collisions for VPCs and subnets.

Authentication events—Cognito in AWS, Cloud Run in GCP, and Azure Functions—are logged in distinct systems (CloudWatch, Cloud Logging, Application Insights, respectively). Without a unified observability layer (e.g., OpenTelemetry Collector with Loki stack), incident investigation becomes a time-consuming task [2].

Moreover, the actual cost of eliminating infrastructure debt grows exponentially after the third phase of the application lifecycle, as illustrated in Table 3.

Table 3

Description of the Difficulties Encountered [2; 5; 8]

IaC Anti-pattern	Symptoms in Multi-cloud	Root Cause	Mitigation Action
Monolithic state file	Terraform state locked during parallel applies; 20–30 minutes deployment delay	Lack of environment/workspace separation	Split state by domain, store in versioned cloud object storage
"Copy-paste DevOps"	Duplicated modules with inconsistent tags; 3–5 redundant VPCs per provider	Absence of module catalog, no enforced IaC code review	Implement internal Terraform registry, enforce PR review
Forgotten provider version	Resource failures after API auto-updates (e.g., azuread 2.x)	Missing version lock in required_provider	Introduce automatic dependabot-style checks, canary updates in dev environment
Skip-plan-apply	terraform apply -auto-approve used in production pipelines	Time-to-market pressure, manual hotfixes	Enforce blocking policies in CI (Sentinel/OPA), mandatory plan-output visualization
Heroic-Ops	Senior engineers push direct changes; knowledge remains undocumented	Lack of runbooks and pair-rotation practices	Adopt GitOps workflow, automate post-mortems, conduct regular chaos engineering exercises

Thus, orchestration efforts often clash with heterogeneous provider limitations and demand the establishment of a unified control plane. IaC technical debt extends beyond unsupported code to encompass organizational practices (or the lack thereof) that hinder the evolution of multi-cloud platforms.

Conclusion. Multi-cloud strategies offer organizations flexibility, geographical distribution, and resilience against vendor lock-in, but they also significantly increase infrastructure management complexity. The analysis shows that without strict adherence to IaC practices, even mature DevOps teams face critical challenges:

- The absence of unified secrets and access policy management across clouds expands the attack surface. Integrating Zero-Trust IaC templates and automated OPA checks transforms security into a "shift-left" process.
- Disparate API limits and network models among providers cause race conditions during parallel apply operations. Adopting a universal control plane (e.g., Crossplane, Cluster-API) and enforcing idempotent Terraform plans mitigate these risks.
- Monolithic state files, rigid provider versioning, and Heroic-Ops culture emerge as major obstacles to platform evolution. Establishing a clear SDLC for IaC artifacts, maintaining a modular template library, and embedding code review processes reduce this debt at early stages.

The proposed "Best Practice ↔ Pitfall" matrix systematically maps threats to countermeasures, while the Zero-Trust-IaC reference architecture demonstrates their practical integration.

In summary, transitioning to multi-cloud IaC is justified only when cultural (GitOps discipline), procedural (IaC SDLC), and technological (Zero-Trust, CSPM automation) practices are strengthened simultaneously, as demonstrated throughout this work.

References

1. Waseem M. et al. Containerization in Multi-Cloud Environment: Roles, Strategies, Challenges, and Solutions for Effective Implementation. arXiv preprint arXiv:2403.12980. 2024. Vol. 1 (1). pp. 11-35.

2. Patel S. Cloud Security Best Practices: Protecting Your Data in a Multi-Cloud Environment. *International Journal of Novel Research and Development*. 2024. Vol. 9 (11). pp. 219-236.

3. Kaur T. Containers in Multi-Cloud Environments: Benefits, Challenges, and Best Practices. *International Journal of Advanced Research and Emerging Trends.* 2024. Vol. 1 (2). pp.146-156.

4. Alouffi B. et al. A Systematic Literature Review on Cloud Computing Security: Threats and Mitigation Strategies. *IEEE Access*. 2021. Vol. 9. pp. 57792-57807.

5. He Y. et al. A Survey on Zero Trust Architecture: Challenges and Future Trends. *Wireless Communications and Mobile Computing*. 2022. Vol. 1. pp. 1-8.

6. Rani P., Singh S., Singh K. Cloud Computing Security: A Taxonomy, Threat Detection and Mitigation Techniques. *International Journal of Computers and Applications.* 2024. Vol. 46 (5). pp. 348-361.

7. Bayya A. K. Leveraging Advanced Cloud Computing Paradigms to Revolutionize Enterprise Application Infrastructure. *Asian Journal of Mathematics and Computer Research.* 2025. Vol. 32 (1). pp. 133-154.

8. Malviya A., Dwivedi R. K. A Comparative Analysis of Container Orchestration Tools in Cloud Computing.2022 9th International Conference on Computing for Sustainable Global Development (INDIACom). IEEE, 2022. pp. 698-703.

9. Centofanti C. et al. Impact of Power Consumption in Containerized Clouds: A Comprehensive Analysis of Open-Source Power Measurement Tools. *Computer Networks*. 2024. Vol. 245. pp. 1-8.

10. Merlino G. et al. FaaS for IoT: Evolving Serverless towards Deviceless in I/O *Clouds. Future Generation Computer Systems*. 2024. Vol. 154. pp. 189-205.