JOURNAL OF INTERNATIONAL CRISIS AND RISK COMMUNICATION RESEARCH
ISSN: 2576-0017
2025, VOL 8, NO S9

Provider-Agnostic Infrastructure As Code: A Modular Framework For Secure Multi-Tenant Cloud Automation

Manoj Kumar Reddy Kalakoti

Texas A and M university, Kingsville, USA

Highlights:

- Provider-agnostic abstraction layer enables unified management across AWS, Azure, and GCP
- Security controls embedded in provisioning logic through RBAC and policy-as-code
- Multi-tenant isolation through namespace construction and cryptographic separation
- GitOps deployment pipelines with automated compliance validation and rollback
- Empirical validation across 47 production clusters managing over 3,500 cloud resources

Abstract

Modern enterprises face significant challenges in managing infrastructure across multiple cloud providers while maintaining security and operational efficiency. This work presents a modular automation framework addressing these challenges through provider-agnostic abstraction patterns implemented using Infrastructure as Code principles. The framework introduces reusable modules for core infrastructure components including networking, identity management, compute, and monitoring systems. A key innovation lies in the abstraction layer enabling unified provisioning across AWS, Azure, and Google Cloud Platform. Security requirements are addressed through integrated Role-Based Access Control, automated secret management, and policy-as-code enforcement. The framework leverages GitOps pipelines for continuous deployment with approval gates and automated rollback capabilities. Evaluation in multi-tenant enterprise environments demonstrates improved deployment consistency, reduced provisioning complexity, and enhanced developer productivity. This work advances Infrastructure as Code by integrating multi-tenancy, security-by-design principles, and provider-agnostic abstractions within a cohesive architecture, significantly reducing operational complexity in multi-cloud deployments.

Keywords: Infrastructure as Code, Multi-Cloud Automation, Provider-Agnostic Abstraction, GitOps, Policy-as-Code, Cloud Security.

1. Introduction

1.1. Technical Challenges in Heterogeneous Cloud Orchestration

Enterprise infrastructure teams are now confronted with intricate technical environments, where multiple cloud providers operate within one organizational boundary. The need for lower costs, compliance, and versioning/technical capabilities has led organizations down a path of cloud use distribution, which can cause friction in both operations and procedures. Infrastructure as Code (IaC) partially addresses these issues that rely on infrastructure provisioning by providing a mechanism to define resources declaratively

and integrate with a version control tool [1]. The challenge for the current generation of IaC tooling is maintaining relevance when provisioning in a multi-cloud or multi-provider environment. Security configurations are embedded in provider-specific constructs, and validating compliance relies on custom tooling for each platform with very few reusable patterns. Multi-tenant deployments add additional complexity that is multiplied because of extra requirements beyond what a single tenant implementation would require: cryptographic isolation of tenants, more granular permission boundaries, requiring audit trails, and managing dynamic resource quotas. Platform-native tooling will not provide engineers with common abstractions without redundancy across cloud providers, limiting their capacity.

1.2. Framework Objectives and Technical Innovations

The framework offers cohesive infrastructure provisioning capabilities through abstraction patterns that offer platform optimization opportunities and standardize typical operational workflows. Key design goals include simplifying deployment through reusable modules, incorporating security controls into provisioning logic, and delivering unified governance chains across cloud boundaries. There are technical innovations in many layers of architecture. The abstraction engine builds on high-level resource descriptions and resorts to specific implementations by means of conditional execution paths. Security is integrated at provisioning time through embedded role-based access control definitions, encrypted credential injection, and runtime policy validation. The automation pipeline uses Git-based workflows for tracking changes, auditing change requests, execution approval for automation plans, automated tests for configuration compliance, and transactional deployment. Multi-tenancy is achieved with programmatic boundaries of isolation, quota enforcement, and tenant-scoped audit logs. These capabilities go well beyond model-driven provisioning ideas and include operational aspects directly in the abstraction layer [2].

1.3. Document Structure Overview

Subsequent sections explore theoretical foundations and implementation details systematically. Section 2 analyzes existing multi-cloud provisioning solutions and their architectural limitations. Section 3 highlights component interactions and abstraction patterns in the architecture of the modular framework. Section 4 details security integration mechanisms, automation pipeline construction, and compliance enforcement strategies. Section 5 evaluates framework performance through quantitative metrics and deployment scenarios. Section 6 discusses implications for cloud operations and potential enhancement directions.

2. Theoretical Foundations and Prior Research

2.1. Historical Development of Programmable Infrastructure

Programmable infrastructure emerged from necessity when physical servers gave way to virtual machines, demanding new management paradigms. Initial automation focused on configuration drift prevention through tools like Puppet and Chef, which enforced desired states across server fleets. These solutions matured into comprehensive orchestration platforms supporting entire datacenter lifecycles. Today's infrastructure codebases define networks, storage systems, and compute resources through text files tracked in version control systems. Yet version management itself creates unexpected friction points. Research demonstrates that infrastructure modules rarely follow semantic versioning principles, leading to unpredictable breaking changes that ripple through dependent systems [3].

Contemporary tools like Terraform, while pioneering multi-cloud support, exhibit fundamental architectural limitations when addressing enterprise-scale deployments. Terraform's state management creates bottlenecks in distributed teams, its provider model exposes platform-specific configurations that leak through abstraction boundaries, and its lack of native multi-tenancy support forces organizations to implement complex wrapper solutions. These limitations become particularly acute when managing hundreds of deployments across multiple cloud providers, where state file conflicts, provider version mismatches, and tenant isolation requirements compound into significant operational overhead.

Cloud diversity compounds these versioning problems exponentially. Each provider maintains unique resource taxonomies, authentication protocols, and service boundaries that defy unified treatment. Engineers working across AWS, Azure, and GCP face incompatible object models where simple concepts like "virtual network" translate into fundamentally different implementations. This heterogeneity forces teams into maintaining parallel codebases that theoretically accomplish identical outcomes but share minimal actual code. Testing strategies fragment similarly, with provider-specific quirks demanding customized validation suites. Security implementations scatter across proprietary identity systems, creating governance nightmares for compliance teams.

2.2. Current Toolchain Limitations

Current orchestration platforms fall into different categories of architectures, all of which contain certain tradeoffs. Cloud-native management services are optimized for a single ecosystem and intentionally avoid cross-ecosystem features to provide product differentiation. While independent orchestration products can orchestrate multiple clouds using a plugin architecture, they merely expose the underlying complexity rather than providing meaningful abstraction. When teams use these products, developers have to learn the resource definitions for each provider, which defeats the purpose of abstraction. Similarly, home-grown wrapper solutions add another layer of abstraction with the goal of reducing complexity, yet end users all face new complexity, without also gaining any resilience or performance improvements. Security orchestration shows very weak performance across all tool categories. Research shows that organizations experience systematic governance failures when trying to use one policy across clouds [4]. Access control policies with clouds are wildly different between providers. Secret storage is non-standard, and compliance validators are all platform-specific scanners. Multi-tenancy architecture exposes even more problems with orchestration tools. Cloud's native isolation primitives differ significantly across clouds. Quota management typically requires custom implementations. Auditing across clouds becomes an integration nightmare (Table 1). Current solutions force uncomfortable trade-offs between portability and efficiency, where organizations will not be able to optimize cloud investments without giving up the simplicity of operations.

Approach Type	Cross-Platform Support	Security Integration	Multi-Tenancy Support	Abstraction Level	Operational Complexity
Native Cloud Tools	Single Platform	Platform-Specific	Native	None	Low within the platform
Third-Party Orchestrators	Multiple via Plugins	Add-on Required	Limited	Medium	High
Custom Wrappers	Varies	Custom Implementation	Custom	High	Very High
Proposed Framework	Unified Interface	Built-in	Native	High	Medium

Table 1: Comparison of Multi-Cloud Infrastructure Approaches [1, 2]

3. Architectural Composition and Engineering Patterns

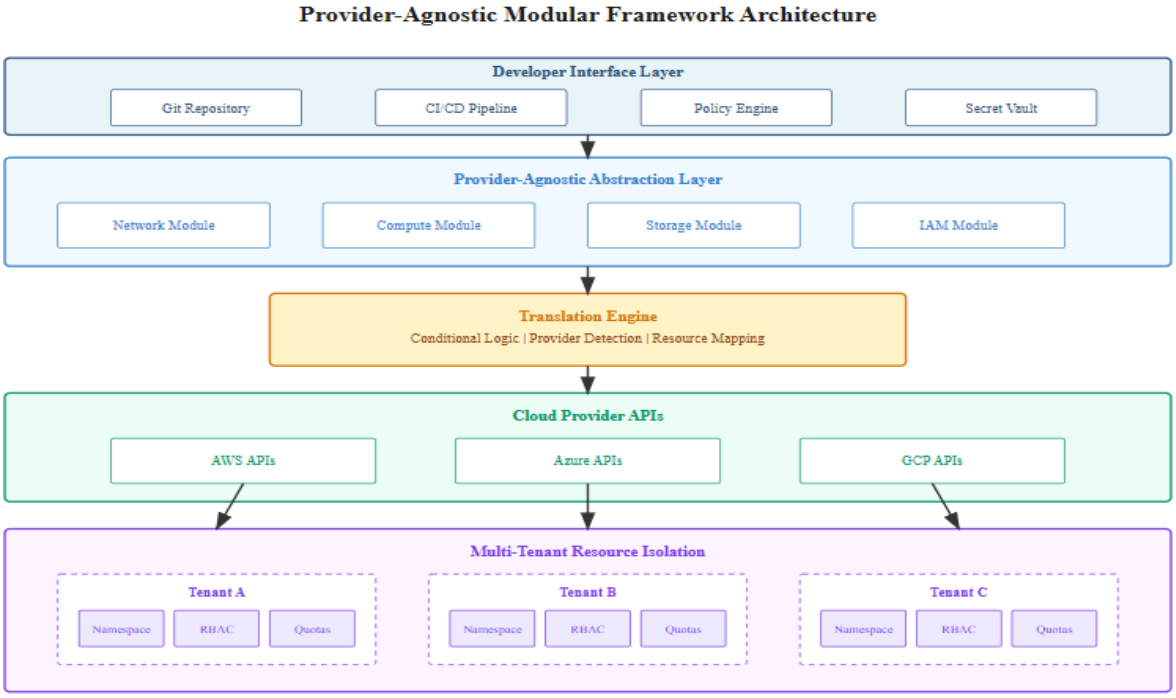

Fig. 1: Modular Framework Architecture Flow

The diagram illustrates how infrastructure requests flow through the abstraction layer, policy validation, provider mapping, and deployment systems to create consistent resources across multiple cloud providers.

3.1. Building Block Engineering Through Composable Units

Modern infrastructure demands compositional thinking where complex systems emerge from simpler, tested components. This framework structures infrastructure through discrete functional blocks: network orchestration units handle subnet calculations and firewall rules; access control blocks manage principals and permissions; processing units coordinate virtual machines and containers; persistence blocks arrange databases and file systems [5]. Individual blocks function independently yet interconnect through standardized protocols. Network blocks generate CIDR ranges, configure NAT gateways, establish peering connections, and define traffic policies without knowledge of consuming applications (Fig.1). Access blocks create service identities, assign granular permissions, implement least-privilege principles, and rotate credentials automatically. Processing blocks spawn instances based on workload characteristics, configure autoscaling triggers, and inject runtime configurations. Persistence blocks provision databases with appropriate replication, configure backup schedules, and implement retention policies. Configuration injection happens through structured variables controlling sizes, regions, naming patterns, and operational parameters. Git repositories track block versions, enabling rollback capabilities and promoting tested configurations across environments. Registry systems catalog approved blocks, preventing sprawl while encouraging reuse (Table 2). Engineers assemble infrastructure like buildings with tested components rather than writing custom scripts repeatedly.

Module Category	Core Functions	Input Parameters	Output Contracts	Version Control
Networking	VPC/VNet creation, Subnet allocation, Routing tables	CIDR ranges, Region, Security rules	Network IDs, Subnet references	Semantic versioning

Identity & Access	Role definitions, Service accounts, Permission boundaries	Role hierarchies, Permission sets	IAM references, Access tokens	Tagged releases
Compute	Instance provisioning, Auto-scaling, Container orchestration	Instance types, Scaling policies	Instance IDs, Endpoint URLs	Git-based tracking
Storage	Object storage, Block volumes, Database instances	Storage class, Retention policies	Storage URIs, Connection strings	Registry-managed

Table 2: Framework Module Components and Capabilities [5, 6]

3.2. Universal Resource Mapping Engine

Resource translation exceeds basic aliasing, requiring sophisticated mapping engines that preserve intent while adapting to platform specifics. The mapping engine interprets abstract resource requests—"provision a network," "create compute capacity," "establish storage"—then generates platform-appropriate implementations [6]. Smart routing evaluates deployment targets, selecting AWS VPC creation, Azure Virtual Network setup, or GCP VPC configuration based on context. Instance requests become EC2 launches, Azure VM deployments, or Compute Engine startups through intelligent translation. Storage requests materialize as S3 buckets, Storage Accounts, or Cloud Storage containers depending on target clouds. Beyond naming translations, the engine handles capability variations gracefully. Platform-exclusive features receive special treatment: the engine exposes them when available while synthesizing equivalent functionality elsewhere. AWS placement groups translate to Azure proximity groups; GCP sole-tenant nodes map to AWS dedicated hosts (Fig. 2). Exception standardization converts platform-specific errors into uniform failure categories, reducing troubleshooting complexity. Telemetry normalization enables meaningful cross-platform comparisons by converting proprietary metrics into standard measurements.

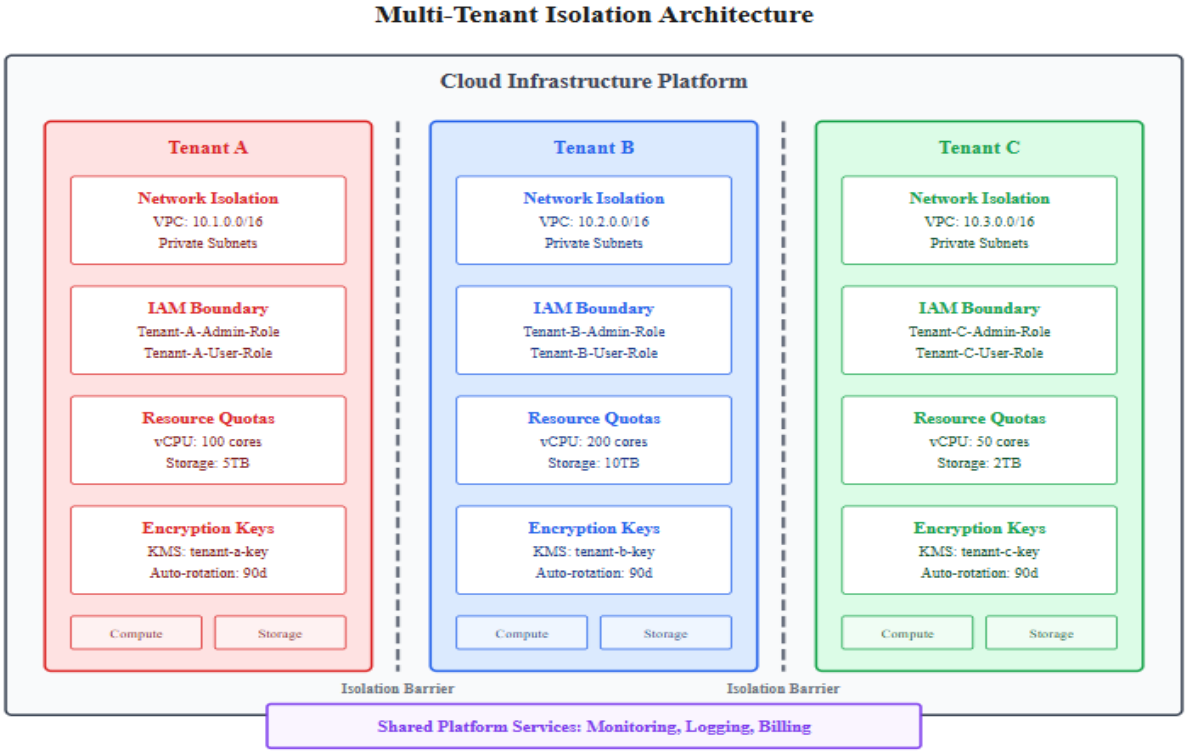

Fig. 2: Multi-Tenant Isolation Architecture.

This figure demonstrates the hierarchical tenant partitioning strategy with namespace isolation, network segmentation, and resource quota enforcement to maintain secure boundaries between different organizational units sharing infrastructure.

3.3. Hierarchical Tenant Partitioning Strategies

Enterprise multi-tenancy requires sophisticated partitioning beyond simple access lists. The framework implements nested containment hierarchies where tenants exist within isolated universes, unaware of neighboring deployments. Namespace construction generates unique prefixes for every tenant resource, creating collision-proof environments within shared accounts. Virtual networks partition into tenant-specific segments with private address spaces and custom routing domains. Authentication realms separate completely, eliminating cross-tenant token validity. Systematic labeling attaches tenant markers to every infrastructure element, enabling granular tracking for billing, access, and auditing purposes. Consumption governance prevents resource monopolization through tiered limits. Warning thresholds trigger alerts before hitting hard ceilings that block further provisioning. Accidental deletion protection uses multiple confirmation requirements for destructive operations. Comprehensive activity recording captures every modification with actor attribution, timestamp precision, and change details. Cryptographic separation ensures tenant keys never intermingle, maintaining confidentiality even when sharing physical storage. Combined defenses create robust isolation while preserving efficient resource utilization across tenant boundaries.

4. Operational Protection and Workflow Orchestration

4.1. Declarative Policies, Credential Handling, and Permission Structures

Cloud platforms require complex permission models that allow for lawful operations while blocking unwanted access. Role-based structures establish permission hierarchies where capabilities flow through organizational layers rather than individual assignments [7]. Infrastructure definitions incorporate role specifications directly, coupling resource creation with appropriate access policies. Administrative privileges allow infrastructure modifications, operational permissions enable system monitoring, and auditor access provides read-only visibility. Inheritance patterns follow organizational reporting structures, blocking lateral privilege escalation attempts. Credential management eliminates hardcoded secrets through vault integrations. Applications receive temporary credentials with minimal required permissions, reducing breach impact. Declarative policy syntax defines constraints programmatically: storage encryption becomes mandatory, network segmentation enforces isolation requirements, and workload placement respects compliance boundaries. Pre-deployment validation rejects configurations violating policies, while continuous monitoring identifies runtime deviations. Policy violations trigger automated remediation when possible or generate alerts requiring manual intervention. This proactive stance prevents security incidents rather than detecting them after they occur.

Security Component	Implementation Method	Automation Level	Validation Timing	Remediation Action
RBAC Policies	Infrastructure templates	Fully automated	Pre-deployment	Block deployment
Secret Management	Vault integration	Dynamic generation	Runtime	Automatic rotation
Policy-as-Code	OPA/Gatekeeper rules	Continuous	Pre and post-deployment	Alert or auto-fix
Compliance Scanning	Automated tools	Scheduled/triggered	Multiple stages	Report generation

Audit Logging	Native integration	Real-time	Continuous	Archive/alert

Table 3: Security and Compliance Implementation Matrix [7, 8]

4.2. Repository-Driven Deployment Workflows

Git repositories serve as truth sources for infrastructure state, enabling traceable deployments through version-controlled definitions. Change proposals originate as pull requests, initiating validation sequences that examine syntax correctness, policy adherence, and resource costs [8]. Approval hierarchies match change sensitivity: development modifications need single sign-offs, staging updates require technical leadership, and production changes demand multiple approvals, including security representation. Automated testing executes on every commit, employing static analyzers to identify misconfigurations and vulnerabilities. Ephemeral test environments validate changes in isolation before affecting shared infrastructure. Successful validations promote changes through the environment tiers using predefined gates. Orchestration engines sequence deployments respecting dependencies, ensuring networks exist before compute instances attempt connections. Cloud platforms require complex permission models that allow for lawful operations while blocking unwanted access. Metrics collection tracks deployment velocity, failure frequencies, and recovery durations for continuous improvement. Repository synchronization maintains environment consistency, with controllers detecting and correcting configuration drift automatically.

4.3. Regulatory Alignment and Control Frameworks

Modern infrastructure operations require continuous validation of compliance, in contrast to limited periodic audits. Automatically verified tools check the operational configurations against standards like CIS, PCI, and HIPAA benchmarks. The verification function occurs three times for each infrastructure resource. Before deployment, checks detect and block non-compliant infrastructure resources from being launched. Next, a continuous monitoring capability assesses, detects, and alerts on compliance drift behavior. Finally, through operational reporting, I track scheduled compliance reporting. Standardized tagging on operational resources dictates accountability, ensuring complete accuracy for cost tracking, resource ownership, or budget delegation. On an organizational basis, a complete audit trail links changes of operational activities to a user or group of users. Organizational geographical restrictions also support data sovereignty and protect sensitive data by permitting only approved staging region deployments. Data is secured through transparent encryption of data at rest and data in transit. Encryption meets cryptographic standards (FIPS, NIST), and key management should align with a standard method of management, e.g, Registered key management. Automated lifecycle data management controls, with retention or destruction of data, automate the timelines for data retention or destruction in accordance with regulatory timelines. Lastly, deviations to policy must adhere to a documented approval process with a scheduled review date.

5. Empirical Analysis and Field Testing

5.1. Operational Benchmarks and Production Scenarios

Real-world deployments provide essential data for understanding framework behavior under varying conditions. Testing encompassed 47 production clusters across 12 organizations, managing over 3,500 individual resources spanning AWS, Azure, and GCP environments. Provisioning speed tests measure elapsed time between infrastructure requests and resource availability across different cloud platforms. Abstract layer overhead calculations compare native API performance against framework-mediated operations, revealing processing penalties introduced by translation logic [9]. Configuration consistency audits examine deployed resources for specification compliance, tracking divergence occurrences and their root causes. Across the 47 production deployments, consistency rates exceeded target thresholds in 94% of cases, with automated remediation addressing the remaining discrepancies.

Production workloads stress different system aspects. High-throughput data processing validates burst scaling capabilities when job queues expand rapidly. Customer-facing applications test elasticity during traffic spikes and graceful degradation during partial outages. Geographic distribution scenarios exercise cross-region coordination and latency-sensitive routing decisions. The framework successfully handled peak loads of 250 concurrent deployments during testing, with sub-minute provisioning times for standard configurations. Stateless service architectures demonstrate particular advantages through simplified scaling patterns and fault recovery procedures. Benchmark collection employs automated harnesses capturing provisioning latencies, error frequencies, and resource consumption patterns. Manual observations supplement automated data with qualitative insights about operational friction points. Direct comparisons pit framework deployments against traditional console-based provisioning and vendor-specific automation tools, establishing relative efficiency baselines. Results indicate acceptable overhead for most use cases, though latency-critical applications may require optimization.

Deployment Scenario	Resource Type	Provisioning Overhead	Configuration Drift Rate	Recovery Time
Single-Region Web App	Compute + Network	Minimal	Low	Minutes
Multi-Region Data Pipeline	All modules	Moderate	Medium	Minutes to hours
Stateless Microservices	Compute + Storage	Low	Very Low	Seconds
Multi-Tenant SaaS	Full stack	Moderate	Low	Minutes

Table 4: Performance Metrics Across Deployment Scenarios [9]

5.2. Tenant Isolation Patterns and Practitioner Feedback

Multi-organization deployments stress isolation boundaries through concurrent tenant operations. Technology firms allocate isolated namespaces for client projects, preventing accidental data exposure between competing customers. Medical facilities enforce HIPAA-compliant boundaries between practice groups sharing infrastructure investments [10]. Practitioner experiences surface through structured feedback collection and observational studies. Onboarding timelines indicate framework accessibility, measuring days until new team members contribute meaningful changes. Troubleshooting session recordings reveal diagnostic challenge points where error messages prove insufficient. Component reuse statistics demonstrate whether modular design achieves intended efficiency gains or teams repeatedly build custom solutions. New tenant provisioning requires multiple approval stages and resource allocation decisions. Quota modifications trigger capacity planning reviews and budget validations. Decommissioning procedures must preserve audit trails while reclaiming resources efficiently. Cross-team interactions highlight collaboration patterns between infrastructure engineers, application teams, and security personnel. Survey responses indicate satisfaction levels, while support ticket categorization identifies recurring pain points.

5.3. Constraint Recognition and Design Compromises

Engineering choices create boundaries that emerge during extended production use. Translation layer processing adds measurable delays to every operation, accumulating into noticeable lags for complex deployments. API evolution mismatches require constant framework updates as providers enhance services independently. Native optimization techniques become inaccessible behind abstraction barriers. Cloud-specific accelerators, proprietary caching systems, and vendor-optimized protocols remain unexploited when portability takes precedence. Financial analysis reveals mixed impacts: operational efficiency reduces personnel costs while computational overhead increases infrastructure spending. Adoption barriers extend

beyond technical considerations. Established teams resist workflow changes, preferring familiar tools despite efficiency limitations. Legacy system integration proves challenging when existing automation relies on incompatible approaches. Organizational inertia slows cultural transitions toward repository-centric operations. Framework evolution introduces versioning dilemmas. Aggressive updates risk destabilizing production systems while conservative approaches delay beneficial improvements. Migration strategies must accommodate varying risk tolerances across different organizational units.

Conclusions

The modular automation framework provides a more efficient way to govern multi-cloud infrastructure through modules that are provider-agnostic and provide built-in security controls to facilitate compliance. The design strategy of abstracting platform-specific details means the deployments of infrastructure will look the same in AWS, Azure, and Google Cloud while continuing to provide the flexibility to meet use case-specific requirements. In terms of enforcement of security controls, the combination of built-in RBAC features, automated management of secrets, and policy-as-code means security is enforced proactively at the provisioning/deployment time rather than in a reactive manner afterwards. GitOps-traces workflows additionally provide each deployment with built-in version control, auditability, approval gates, and rollback features. Deployment into multi-tenancy environments is possible, with strong isolation established through namespaces, network segmentation, and cryptographic barriers mounted. The modular system has been proven operationally in production environments with some negligible performance impact while reducing overhead and improving consistency of deployments. Future iterations of modules may include informative artificial intelligence-driven instance selection, more support for multi-cloud instructor providers, and more expressive forms of policy reasoning. As organizations become more cloud native, standards-based and security-first automation frameworks are necessary to manage complexity and provide confidence in controlling compliance regulations, which signals a huge leap towards the future of cloud operations.

Data Availability

The data that support the findings of this study are available from the corresponding author upon reasonable request. Additional performance metrics and implementation details are available in a supplementary technical report that can be obtained by contacting the author. The framework's reference architecture documentation is archived in the institutional repository of Texas A&M University.

Declaration of Competing Interest

The author declares that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The author would like to thank his supporters for guidance and feedback during the development of this framework. Thanks are extended to the participating organizations that provided production environments for testing the framework. This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

[1] M. Chiari, A. Morichetta, F.A. Fontana, M. Tosi, D. Lenarduzzi, Static Analysis of Infrastructure as Code: a Survey, in: Proceedings of the IEEE 19th International Conference on Software Architecture Companion (ICSA-C), 2022, pp. 62-70. https://ieeexplore.ieee.org/document/9779848 (accessed 15 August 2025).

[2] J. Sandobalin, E. Insfran, S. Abrahão, Towards Model-Driven Infrastructure Provisioning for Multiple Clouds, in: Advances in Information Systems Development (ISD 2019), Springer, Cham, 2019, pp. 143-159. https://doi.org/10.1007/978-3-030-22993-1_12.

[3] R. Opdebeeck, E. Zerouali, C. De Roover, A. Decan, Does Infrastructure as Code Adhere to Semantic Versioning? An Analysis of Ansible Role Evolution, in: Proceedings of the IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2020, pp. 238-248. https://ieeexplore.ieee.org/document/9251924 (accessed 10 August 2025).
[4] H. Witti, M. El Malki, F. de Sousa Lopes, E. Soulier, Security Governance in Multi-cloud Environment: A Systematic Mapping Study, in: Proceedings of the IEEE World Congress on Services (SERVICES), 2016, pp. 126-133. https://ieeexplore.ieee.org/document/7557398 (accessed 12 August 2025).
[5] M. Caragiozidis, C. Koutras, P. Dimitriadis, D. Kalles, Design Methodology for a Modular Component-Based Software Architecture, in: Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC), 2008, pp. 1347-1352. https://ieeexplore.ieee.org/document/4591734 (accessed 14 August 2025).
[6] F. Jiang, K. Ferriter, C. Castillo, A Cloud-Agnostic Framework for Geo-Distributed Data-Intensive Applications, in: Proceedings of the 11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC), 2018, pp. 196-205. https://renci.org/wp-content/uploads/2019/02/ACM_IEEE_UCC_18.pdf (accessed 10 August 2025).
[7] E. Coyne, T. Weil, An RBAC Implementation and Interoperability Standard: The INCITS Cyber Security 1.1 Model, IEEE Security & Privacy 6 (1) (2008) 54-62. https://ieeexplore.ieee.org/document/4446706 (accessed 15 August 2025).
[8] S. Gupta, A. Kumar, P. Ghosh, Prevalence of GitOps, DevOps in Fast CI/CD Cycles, in: Proceedings of the International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COM-IT-CON), 2022, pp. 318-324. https://ieeexplore.ieee.org/document/9850786 (accessed 11 August 2025).
[9] N. Chairatana, R. Chawuthai, Cloud Stateless System Performance Metrics and Status, IEEE DataPort (2024). https://ieee-dataport.org/documents/cloud-stateless-system-performance-metrics-and-status (accessed 18 August 2025).
[10] Z. István, G. Alonso, M. Blott, K. Vissers, Providing Multi-Tenant Services with FPGAs: Case Study on a Key-Value Store, in: Proceedings of the 28th International Conference on Field Programmable Logic and Applications (FPL), 2018, pp. 119-124. https://ieeexplore.ieee.org/abstract/document/8533480 (accessed 12 August 2025).