The American Journal of Engineering and Technology
ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 88-95
DOI 10.37547/tajet/Volume07Issue07-10

Check for updates

Efficiency of Terraform and Kubernetes Integration in DevOps Practices

Nikita Romm
Senior Staff DevOps Engineer, Palo Alto Networks Tel Aviv, Israel

OPEN ACCESS
SUBMITTED 09 June 2025
ACCEPTED 17 June 2025
PUBLISHED 22 July 2025
VOLUME Vol.07 Issue 07 2025

CITATION

Nikita Romm. (2025). Efficiency of Terraform and Kubernetes Integration in DevOps Practices. The American Journal of Engineering and Technology, 7(07), 88–95. https://doi.org/10.37547/tajet/Volume07Issue07-10

Abstract: This article examines the effectiveness of combining Terraform and Kubernetes within DevOps workflows. Against the backdrop of microservices architectures and cloud-native environments, the synergy between Infrastructure as Code (IaC) and container orchestration has become increasingly important. Our contribution lies in systematically exploring how Terraform and Kubernetes can be used together during provisioning, CI/CD pipelines, and autoscaling. We compare their feature sets, review real-world cluster-deployment case studies, and discuss state-management strategies and self-healing mechanisms. Key recommendations cover modular infrastructure design, clear separation of responsibilities, and adoption of GitOps principles. Drawing on official documentation, English-language vendor publications, and industry reports, our analysis identifies the integration's benefits for faster application delivery, higher system stability, and repeatable processes. We employ comparative documentation review, content analysis of DevOps community resources, and case-study methodology. Practical guidance for optimizing Terraform–Kubernetes collaboration concludes the paper. These insights will be valuable to DevOps engineers, architects, and deployment-automation specialists, reflecting current industry trends and laying groundwork for future research.

Keywords: Terraform, Kubernetes, DevOps, Infrastructure as Code, container orchestration, CI/CD automation, provisioning, autoscaling, GitOps, modular infrastructure

Introduction

Modern DevOps practices focus on automating and accelerating application development, deployment, and scaling. In this context, Infrastructure as Code (IaC) and container orchestration have emerged as cornerstone technologies. Terraform and Kubernetes are two widely adopted open-source tools in the DevOps toolkit. Terraform, developed by HashiCorp, enables teams to declare and provision infrastructure resources—virtual machines, networks, databases, and more—using a declarative configuration language [3]. Kubernetes (K8s) provides a robust orchestration platform for containerized workloads, automating deployment, scaling, and management of containers [1]. Both projects launched in the mid-2010s and by 2025 had become de-facto standards—Terraform for multi-cloud infrastructure management and Kubernetes for container platforms. Yet they address distinct layers of the stack, raising the question: how effective is it to combine Terraform with Kubernetes in DevOps workflows?

The purpose of this article is to examine the benefits and patterns of integrating Terraform and Kubernetes within DevOps, particularly for infrastructure automation, CI/CD pipeline implementation, and application scalability. Specifically, we will explore:

- the respective roles and typical use cases of Terraform and Kubernetes;
- real-world integration scenarios—from provisioning Kubernetes clusters via Terraform to deploying applications on those clusters;
- how this combined approach impacts DevOps metrics (deployment velocity, system stability, reproducibility, etc.);
- best practices and potential pitfalls.

This topic is highly relevant because the Terraform+Kubernetes pairing unifies infrastructure and application management—an essential capability when working with microservices architectures and cloud-native, horizontally scalable systems.

Materials and Methods

This study draws on a range of sources: official HashiCorp and CNCF documentation, technical whitepapers, industry analyses, and hands-on case studies from practicing engineers. Key references include the HashiCorp guide to the Terraform Kubernetes provider [6], comparative blog posts from Spacelift, DuploCloud, and ControlPlane examining Terraform vs. Kubernetes approaches [2], as well as community-generated content (Medium articles, forum discussions) [1, 3–5].

Methodologically, we conducted a comparative feature analysis to map the strengths and constraints of both tools. In parallel, we performed case-study analyses of concrete integration patterns—such as spinning up an EKS cluster with Terraform and then deploying microservices on it via Kubernetes—to assess their effects on CI/CD speed and reliability. The findings are synthesized into high-level insights and actionable recommendations, all substantiated by citations to authoritative sources.

Results

Before examining their joint usage, it is essential to clarify each tool's individual role. Terraform excels at the declarative provisioning and lifecycle management of infrastructure "as code" across a variety of environments—from public clouds (AWS, Azure, GCP) to on-premises resources [2]. DevOps engineers use Terraform to automate the creation of virtual machines, network configurations, database clusters, and other foundational components that applications depend on. With support for hundreds of providers, Terraform can stand up complex infrastructures in minutes, eliminating manual setup. Its key strengths include multi-cloud capability, a persistent state file, built-in plan/apply workflows (e.g. `terraform plan` previews changes), and automatic resource dependency management [6]. In short, terraform answers: "What infrastructure must we create, and how can we reproducibly build it?"

By contrast, Kubernetes tackles the question: "How do we deploy and keep our containerized application running reliably?" As a container orchestration layer, Kubernetes ingests declarative YAML manifests describing the desired state of an application—number of replicas, container images, service endpoints, and so on—and relentlessly works to maintain that state. It schedules containers across cluster nodes, monitors health (restarting failed pods), scales workloads in response to demand, and provides built-in services like load balancing, service discovery, configuration maps, and secrets management [1]. Kubernetes shines in microservices architectures, where dozens or hundreds of containers must interoperate and scale seamlessly. In modern CI/CD pipelines, Kubernetes enables rolling updates so new application versions can be deployed without downtime.

Put simply, Terraform and Kubernetes operate at different layers: Terraform manages external infrastructure (for example, provisioning a Kubernetes cluster in the cloud), while Kubernetes manages the workloads running within that infrastructure—the containers themselves. Though often compared as "Terraform vs. Kubernetes," they are in fact complementary: Terraform is best suited for low-level resource provisioning (VMs, networks, load balancers), and Kubernetes for application lifecycle management (container deployment, scaling, and health) [2].

Below is a concise side-by-side comparison of Terraform and Kubernetes along their core dimensions.

Table 1. Comparing Terraform and Kubernetes by Key Characteristics [1]

Criterion	**Terraform (IaC)**	**Kubernetes (Orchestration)**
Primary Use Case	Managing infrastructure—VMs, networks, databases, load-balancers, etc. (Infrastructure as Code)	Managing containerized applications—their deployment, scaling and rolling updates
Configuration	Declarative HCL templates; maintains a persistent state file of infrastructure changes	Declarative YAML manifests; continuously reconciles desired vs. actual cluster state
Core Focus	Creating, modifying or destroying resources according to plan (multi-cloud provisioning)	Orchestrating containers within the cluster (self-healing, autoscaling)
Scaling	Scales infrastructure (e.g. adds nodes/VMs) by updating configuration	Automatically scales application replicas based on runtime metrics
Self-Healing	No built-in recovery—relies on redundancy (Terraform won't restart a failed VM)	Yes—restarts crashed containers, recreates failed pods
Multi-Platform	Yes—supports any cloud or on-premises provider via plugins	Yes—can run on any cloud or on-premises environment
State Management	Requires a state file to track infrastructure across runs	Stores cluster state in etcd and continually reconciles desired vs. current state
Scope	Entire infrastructure stack—from network policies to Kubernetes clusters	Containers and their associated resources within an existing cluster

From this comparison, it's clear that Terraform and Kubernetes address different layers of the stack—even while overlapping at the cluster boundary. Terraform handles broad, multi-cloud infrastructure concerns, whereas Kubernetes dives deep into application-level orchestration. Used together, they cover the full lifecycle: Terraform provisions the cluster—and Kubernetes runs your workloads on it.

In modern DevOps, it's increasingly common to pair Terraform and Kubernetes so that each tool complements the other. Below are the primary integration scenarios and their benefits.

One of the most straightforward patterns is to use Terraform to provision the Kubernetes cluster itself.

Virtually every major cloud provider offers a managed Kubernetes service—EKS on AWS, AKS on Azure, GKE on GCP—and Terraform provides a dedicated provider for each. In practice, a few Terraform blocks can describe how many control-plane and worker nodes you need, their instance types and networking settings—and in minutes stand up a full cluster. Once the cluster exists, the same Terraform code can invoke its Kubernetes provider to configure in-cluster resources (namespaces, RBAC roles, initial Deployments) via the Kubernetes API. According to HashiCorp's documentation, the Terraform Kubernetes provider serves as a bridge that allows Terraform to manage Kubernetes objects alongside other infrastructure [5]. This unified approach lets you describe your entire stack—from VMs to Pods—in a single declarative language (HCL). For example, a Terraform run can sequentially: create EC2 instances, provision an EKS cluster on them, and then deploy your application into that cluster exactly as if you had run kubectl apply, but all driven by Terraform [6]. DevOps teams love this workflow because they never have to switch contexts or maintain separate manifests—everything lives in one version-controlled codebase.

That consistency pays off in repeatable, drift-free environments. Running the identical Terraform configuration against dev, staging and prod yields the same infrastructure and cluster setup every time [3], drastically reducing "works on my machine" issues and human error. In a CI/CD pipeline, you can automatically run terraform plan and terraform apply whenever infra code changes, moving closer to a GitOps-style workflow. Researchers have found that managing Kubernetes resources with Terraform not only minimizes ad-hoc YAML edits, but also brings Terraforms stateful dependency graph into the Kubernetes world—ensuring, for instance, that a Namespace is created before its associated Deployment is ever applied [5]. As a result, configuration mistakes drop significantly, since Terraform inherently understands and orders resource dependencies rather than relying on manual orchestration [6].

However, there are also limitations. Experts advise using Terraform judiciously for managing dynamic in-cluster resources. When applications change frequently (for example, daily deployments), running Terraform for every update becomes cumbersome; in such cases, it's more practical to leverage Kubernetes's native tooling—Helm charts or GitOps tools like Argo CD. Guides typically recommend reserving Terraform for relatively static or infrastructure-level components (cluster provisioning, network policies, ingress controllers), while treating business-logic workloads with CI/CD pipelines and Helm releases [3]. In other words, Terraform excels at creating the cluster itself and its foundational configuration, but orchestrating dozens of microservices through one monolithic .tf file is unwieldy. The best practice is a hybrid approach: use Terraform for the "platform layer" and underlying infrastructure, and let Kubernetes-centric pipelines handle application deployments. This pattern preserves the benefits of a single workflow for infrastructure management without sacrificing Kubernetes's agility. Overall, embedding Kubernetes resource management into an existing Terraform workflow feels natural—and becomes the logical next step when your entire infrastructure is already declared as code [5].

The synergy of Terraform and Kubernetes is most evident in CI/CD pipelines, where they automate the full Continuous Integration/Continuous Deployment cycle. A typical microservices deployment pipeline might look like: build container image → run tests → deploy to staging → terraform apply to prepare production infrastructure → kubectl apply to roll out the application in production. In this sequence, Terraform and Kubernetes act in concert. Many teams streamline further by unifying both tools within a single pipeline. As one medium article observes, "By integrating Terraform and Kubernetes into a CI/CD pipeline, teams can automate provisioning infrastructure, configuring K8s clusters, and deploying containerized apps—delivering value faster, more consistently and with fewer errors" [4]. Indeed, with the right configuration, deploying a new service can be as simple as triggering one CI/CD job that spins up everything from the bare VM to the running application.

For example, when you push to the main branch of your infrastructure-as-code repository, GitLab CI can automatically invoke a terraform apply against the production workspace—provisioning or updating everything from the Kubernetes cluster itself to all supporting resources. Immediately afterwards, that same pipeline can call Argo CD or Helm to deploy your application into the freshly configured cluster. The result is a fully launched, scalable deployment with virtually no manual intervention. For DevOps teams, this drastically shortens time-to-market: infrastructure provisioning and application rollout happen automatically and in parallel, yielding more frequent, predictable releases and minimizing configuration errors. Because both infrastructure and application manifests live in Git, any version of the system (infrastructure + code) is fully reproducible, simplifying debugging and rollbacks. HashiCorp highlights Terraform's "full lifecycle management"—not only creating but also updating or destroying resources as needed—which integrates seamlessly into CI/CD workflows [6]. Kubernetes, in turn, guarantees zero-downtime deployments via rolling updates and readiness probes. Together, they fulfill DevOps objectives of fast, reliable updates with minimal effort.

Of course, building such a pipeline demands expertise. The learning curve for Kubernetes and Terraform is steep [1]: engineers must master cloud infrastructure concepts, container orchestration, and both HCL and YAML configurations. Yet pioneers—particularly in fintech—report cutting release cycles from weeks to hours by adopting IaC and Kubernetes orchestration. Reliability also climbs, as identical infrastructure definitions eliminate human drift and automated tests can even validate Terraform plans.

Another critical efficiency gain comes from autoscaling. Kubernetes was designed for horizontal pod autoscaling (HPA), while Terraform—though not inherently dynamic—lets you adjust infrastructure parameters (e.g., node count) by changing a single line in your configuration. In production environments, two-stage scaling is employed: the Horizontal Pod Autoscaler adjusts pod counts in response to CPU and memory metrics; the Kubernetes Cluster Autoscaler evaluates node utilization and dynamically provisions or decommissions nodes to preserve designated cluster capacity. Although many rely on native cloud autoscaling groups, Terraform can manage those too. This declarative approach ensures performance targets are met either at the container layer (via Kubernetes) or the infrastructure layer (via Terraform). The system handles abrupt traffic surges autonomously, ensuring efficient resource usage: the Kubernetes Cluster Autoscaler provisions additional nodes as demand increases, the Horizontal Pod Autoscaler adjusts pod replicas, and the scheduler balances workloads across available capacity. Such deployment remains robust under heavy load in large-scale cloud infrastructures.

Additionally, Terraform enables precise cost control. By defining resources in code, teams can right-size VMs, storage, and other components for each application, revisiting those definitions as requirements evolve. Practitioners note that Terraform-driven provisioning avoids both under- and over-provisioning, trimming infrastructure spend [3]. Kubernetes complements this by efficiently scheduling workloads across available capacity, boosting server utilization. In combination, Terraform and Kubernetes deliver both scalable performance and cost-effective infrastructure.

Below is a concise summary of the key advantages and drawbacks of integrating Terraform with Kubernetes in DevOps workflows.

Table 2. Pros and Cons of Terraform + Kubernetes in DevOps Practices

Advantages of Joint Use	**Drawbacks of Joint Use**
Consistency & Reproducibility. A unified declarative approach to both infrastructure and application ensures identical environments, minimizing configuration errors and human-induced drift. All changes go through version control and CI/CD pipelines.	**High Complexity & Skill Requirements.** Combining two powerful platforms increases stack complexity and demands engineers versed in both Terraform and Kubernetes, raising the entry barrier for smaller teams.
Faster Delivery & CI/CD Agility. Automated provisioning and deployment accelerate development cycles and time-to-market, allowing pipelines to adapt rapidly to evolving business needs.	**Overkill for Simple Projects.** For small or straightforward infrastructures, running both Terraform and Kubernetes can be unnecessarily heavy and cost-inefficient compared to lighter alternatives.
Simplified Management of Complex Systems. Terraform orchestrates resource dependencies	**Potential Resource-Management Conflicts.** Clear boundaries must be drawn between Terraform's

Advantages of Joint Use	Drawbacks of Joint Use
automatically, while Kubernetes provides self-healing and workload monitoring—reducing manual ops work and letting teams focus on features.	infrastructure role and Kubernetes's in-cluster management; otherwise, concurrent changes can desynchronize state and cause conflicts.
Scalability & Reliability. Together they deliver autoscaling at two layers—Kubernetes for pods (HPA) and Terraform for nodes—while Kubernetes's healing features and Terraform's resource provisioning minimize downtime.	**Longer CI/CD Runtime.** Running terraform apply followed by Kubernetes deployments can lengthen pipeline execution time; while parallelization helps, overall processes may still take noticeably longer.
Multi-Cloud Portability. Terraform's provider-agnostic IaC enables identical stacks across clouds or on-premises, and Kubernetes ensures container portability between those environments.	

Despite these clear benefits—consistency, agility, manageability, scalability, and portability—the added complexity and learning curve mean teams should carefully assess their project's size and criticality before adopting both tools in tandem. In practice, Terraform + Kubernetes integration shines in medium to large efforts, where the automation payoff justifies the investment in training and operational overhead.

Discussion

The findings reaffirm that pairing Terraform with Kubernetes elevates DevOps effectiveness—especially in large-scale, fast-moving projects. By treating both infrastructure and applications as code, teams achieve deeper automation and uniform management in line with GitOps principles. Crucially, the integration process must respect each tool's strengths.

Practitioners concur that Terraform and Kubernetes don't compete but complement one another [2]. Terraform handles external resources—VMs, networks, databases, object storage, CDNs—that Kubernetes isn't designed for. Kubernetes, in turn, manages containerized workloads, ensuring self-healing and desired-state enforcement. Together, they span the full stack from hardware provisioning to application runtime.

Below are actionable recommendations to optimize Terraform + Kubernetes workflows, covering modular design, responsibility boundaries, state management, and GitOps adoption.

Table 3. Recommended Practices for Terraform + Kubernetes Integration

Practice	Description & Benefits	Risks & Caveats
Adopt a modular Terraform layout	Encapsulate infrastructure components into reusable modules, standardizing configuration and reducing error rates.	Over-modularization can complicate maintenance and debugging.
Clearly separate responsibilities	Assign Terraform to manage external infra (VMs, networks, clusters) and Kubernetes to handle in-cluster resources (Deployments, Services, ConfigMaps).	Unclear boundaries may lead to state conflicts and make troubleshooting harder.

Practice	Description & Benefits	Risks & Caveats
Store Terraform state in a remote backend	Use remote backends (e.g., Terraform Cloud, S3 + DynamoDB, Azure Storage) for secure, shared state and locking.	Backend outages can block deployments; implement backups and DR procedures.
Limit Terraform's Kubernetes provider to static objects	Restrict Terraform's in-cluster management to stable resources (Namespaces, RBAC policies, NetworkPolicies), avoiding frequent updates to dynamic workloads.	Managing dynamic resources via Terraform can slow CI/CD pipelines and introduce drift.
Embed Terraform + Kubernetes in a GitOps workflow	Store all infra and application manifests in Git, driving deployments via automated pipelines for full traceability and auditability.	Requires strict change-management discipline and PR governance to prevent out-of-band edits.

Implementing these practices builds a robust, scalable DevOps platform: modular Terraform code streamlines reuse; defined responsibility zones prevent resource conflicts; remote state backends bolster reliability; and a unified GitOps approach ensures transparency, version control, and audit trails for every change.

It's worth noting that the introduction of the Kubernetes provider for Terraform has blurred the lines between infrastructure provisioning and orchestration. By speaking directly to the Kubernetes API, Terraform lets DevOps engineers define in-cluster resources using familiar HCL modules—in many cases eliminating the need to hand-write dozens of YAML manifests. However, as Spacelift's documentation cautions, this isn't a silver bullet: "avoid using Terraform to manage in-cluster K8s resources that change frequently; use Helm or Kustomize for that" [3]. In other words, adopting a hybrid approach doesn't render Kubernetes-native tools obsolete—Helm charts, Operators, and CI/CD pipelines remain essential. Terraform excels at unifying and automating across layers, but you shouldn't overload it with tasks better suited to Kubernetes itself. Sticking to these best practices helps teams reap the benefits without introducing unnecessary complexity.

Security and secrets management deserve special attention when combining Terraform and Kubernetes. Store access credentials—such as kubeconfig files for the Terraform provider or cloud API keys—in a dedicated secret store (Vault, KMS), and grant Terraform only the minimum required privileges. Remember, Terraform state can contain sensitive information, so keep it in an encrypted remote backend with strict access controls. On the Kubernetes side, you'll still need to manage RBAC policies—Terraform can automate those too, but plan your workflows so that you first provision the cluster, then apply role and permission changes, ensuring your state remains consistent and secure.

Looking ahead, we're already seeing even deeper integrations: Kubernetes operators that trigger Terraform runs in response to cluster events, effectively turning Kubernetes into an infrastructure controller (though these remain niche). The rise of Platform Engineering is another trend: teams are building internal developer platforms on top of Kubernetes, with Terraform acting as the backend for "infrastructure on demand." All signs point to Terraform and Kubernetes continuing to coexist and evolve together. HashiCorp actively maintains the Kubernetes provider, and the DevOps community is refining patterns for when to lean on each tool.

Ultimately, both theoretical analyses and real-world case studies confirm the effectiveness of the Terraform + Kubernetes combination. Organizations benefit from faster, more consistent deployments, flexible scaling, and built-in resilience. By following sound architectural guidance, teams can create truly scalable DevOps workflows—from commit to production—that are fully automated, transparent, and under version control. That, at its core, is the essence of DevOps as both a culture and a practice.

Conclusion

Our analysis demonstrates that combining Terraform and Kubernetes in DevOps workflows offers a powerful solution for automating infrastructure and CI/CD pipelines in modern cloud-native applications. Terraform lays down an "infrastructure as code" foundation—provisioning all required resources, including the Kubernetes cluster itself, in a consistent manner across development, staging, and production environments. Kubernetes then takes over container orchestration, ensuring application reliability and on-demand scaling.

Together, these tools deliver key benefits:

- Rapid, reliable deployments: Unified, declarative configurations eliminate manual steps and reduce deployment errors.
- Elastic scalability: Kubernetes' pod autoscaling and Terraform-driven infrastructure scaling adapt capacity to real-time demand.
- Predictable, repeatable environments: Infrastructure-as-code plus containerized workloads guarantee identical behavior from local dev machines through production.
- Multi-cloud portability: Terraform configurations and Kubernetes manifests can be applied to any cloud or on-premise platform.

In practice, adopting Terraform + Kubernetes requires initial investment in pipeline design and team training, but pays dividends in faster release cycles, fewer incidents, and more efficient resource utilization. Organizations embracing these technologies report significant reductions in deployment lead time and operational toil—benefits that far outweigh the onboarding cost for medium and large projects with dynamic requirements.

DevOps engineers and architects can leverage our findings when selecting tools for continuous delivery. It is recommended:

1. Terraform for provisioning infrastructure (VMs, networking, managed Kubernetes clusters).
2. Kubernetes for deploying, scaling, and self-healing containerized applications.
3. CI/CD integration that invokes Terraform for infra changes and Kubernetes tools (Helm, kubectl, ArgoCD) for app rollouts.
4. Best practices such as remote state storage, clear separation of responsibilities between Terraform and Kubernetes, modular Terraform code, and dedicated workspaces per environment.

Together, Terraform and Kubernetes embody the "everything as code" principle—turning complex deployments into automated, auditable processes. By eliminating repetitive manual tasks, speeding feedback loops, and raising reliability, this pairing has emerged as one of the most effective DevOps toolchains available today—and industry experience suggests it will remain a cornerstone of cloud-native delivery for years to come.

References

1. Fahim, Marium. Terraform Vs Kubernetes – Selecting The Right Tool For Your System. – URL: https://cyberpanel.net/blog/terraform-vs-kubernetes (Accessed: 02.05.2025). – Text : electronic.
2. Gaydos, Bob. Terraform vs. Kubernetes: Choosing the Right Tool for Platform Engineering. – URL: https://duplocloud.com/blog/terraform-vs-kubernetes-choosing-the-right-tool-for-platform-engineering/ (Accessed: 07.05.2025). – Text : electronic.
3. Hashem, Faisal. How to Deploy Kubernetes Resources with Terraform. – 2024. – URL: https://spacelift.io/blog/terraform-kubernetes-deployment (Accessed: 28.04.2025). – Text : electronic.
4. How to use Terraform and Kubernetes in DevOps. – 2024. – URL: https://medium.com/@alexeusgr/terraform-and-kubernetes-in-devops-3a3db6ec0db1 (date of access: 02.05.2025). – Text : electronic.
5. Katz, Eyal. Orchestrating Kubernetes with Terraform: A Step-by-Step Guide to Building Your Container Empire. – 2023. – URL: https://controlplane.com/community-blog/post/orchestrating-kubernetes-with-terraform (date of access: 25.04.2025). – Text : electronic.
6. Manage Kubernetes resources via Terraform. – URL: https://developer.hashicorp.com/terraform/tutorials/kubernetes/kubernetes-provider (accessed: 30.04.2025). – Text: electronic.