International Journal of Creative and Open Research in Engineering and Management
ISSN: **3108-1754 (Online)**
Volume 02 Issue 05 May-2026 | Impact Factor: 3.5

Devops Automation Pipeline Deployment using Infrastructure as Code (Iac)

Dhanush V[1], Vishal G[1], Valarselvi N[1], Ezhulmalai S[1], Saranya T[1]
[1]UG Students, Department of Computer Science and Engineering
Supervised by**: Prof. A. Kannammal**
M.E, Ph.D., Department of Information Technology
Jayalakshmi Institute of Technology, Thoppur, Dharmapuri – 636352, Tamil Nadu, India
{610922104014, 610922104107, 610922104101, 610922104021, 610922104080}@jit.ac.in

How to Cite this Article:

V, D., G, V., N, V., S, E. & T, S. (2026). Devops Automation Pipeline Deployment using Infrastructure as Code (Iac). International Journal of Creative and Open Research in Engineering and Management, 02(05). https://doi.org/10.55041/ijcope.v2i5.125

https://doi.org/10.55041/ijcope.v2i5.125

Abstract—The rapid adoption of cloud-native application delivery has intensified the need for automated, reliable, and scalable software deployment solutions. Conventional software deployment methods rely heavily on manual processes, leading to configuration inconsistencies, deployment failures, high operational costs, and slower release cycles. To overcome these challenges, this paper proposes a DevOps Automation Pipeline integrated with Infrastructure as Code (IaC) that automates the complete application lifecycle from code commit to production deployment. The proposed system employs GitHub for source code management, Azure DevOps Pipelines for Continuous Integration and Continuous Deployment (CI/CD), and Terraform for declarative cloud resource provisioning on Microsoft Azure. Containerization using Docker and orchestration with Kubernetes enhance application portability, scalability, and environment consistency. Azure Monitor and Log Analytics provide real-time observability into pipeline execution and deployed application health. Experimental validation demonstrates substantial improvements over manual baselines: 95% reduction in deployment lead time (from days to under 18 minutes), 80% reduction in change failure rate, and 90% reduction in mean time to recovery.

Keywords—DevOps; Infrastructure as Code; CI/CD Pipeline; Terraform; Microsoft Azure; Docker; Kubernetes; GitHub Actions; Azure DevOps; Continuous Integration; Continuous Deployment; Cloud Automation.

I. INTRODUCTION

The contemporary software development landscape demands faster, more reliable, and scalable application delivery. DevOps — a cultural and technical paradigm bridging software development and IT operations — promotes collaboration, automation, continuous integration, and continuous delivery, transforming how organizations develop, test, and deploy systems [1].

Infrastructure as Code (IaC) is a cornerstone of modern DevOps. It refers to the management and provisioning of computing infrastructure through machine-readable configuration files, enabling infrastructure to be version-controlled, reviewed, tested, and deployed automatically — eliminating configuration drift and enabling consistent, repeatable environment setup [2].

Traditional software delivery relied on manual server configuration, deployment script execution, and environment verification. This introduced risks including configuration mismatches between development, staging, and production environments; human errors during deployments; slow release cycles; and high operational overhead — challenges that become unsustainable in microservices and multi-cloud scenarios [3].

This paper presents the design, implementation, and validation of a DevOps Automation Pipeline using IaC. The system employs GitHub for source code management, Azure DevOps Pipelines for CI/CD orchestration, Terraform for cloud infrastructure provisioning on Microsoft Azure, and Docker and Kubernetes for containerization and orchestration. The pipeline automates every delivery stage without human intervention.

The remainder of this paper is structured as follows: Section II reviews related literature. Section III describes system objectives and architecture. Section IV details system specifications. Section V presents system design. Section VI discusses implementation modules. Section VII presents results and performance benchmarks. Section VIII outlines applications, and Section IX concludes with future work directions.

II. LITERATURE REVIEW

A. Survey on CI/CD Practices and DevOps Automation

Humble and Farley [2] introduced the concept of deployment pipelines in their seminal work on continuous delivery, establishing foundational concepts of build automation, automated testing, and environment management. Leppanen et al. [3] examined CI/CD adoption in large organizations, finding up to 200× more frequent deployments and 24× faster failure recovery compared to low-performing teams.

The DORA State of DevOps Reports (2018–2023) demonstrate that elite DevOps performers deploy thousands of times per day with lead times measured in hours [4]. Zhao et al. [5] confirmed that projects adopting CI demonstrate higher commit frequencies and faster defect resolution. Shahin et al. [6] identified four key CI/CD adoption challenges: pipeline configuration complexity, test maintenance, infrastructure provisioning delays, and security integration — three of which the proposed system directly addresses.

B. IaC Tools, Frameworks, and Cloud Automation

Morris [7] provides a comprehensive IaC framework distinguishing mutable versus immutable infrastructure and declarative versus imperative approaches — concepts directly relevant to Terraform usage in this project. Rahman et al. [8] identified anti-patterns in Terraform scripts and established quality guidelines followed here.

Burns et al. [9] describe the architectural evolution of Kubernetes, emphasizing declarative configuration and self-healing. Merkel [10] demonstrated Docker containers as lightweight portable execution environments that eliminate the 'works on my machine' problem. Hilton et al. [11] confirmed that CI-adopting projects demonstrate measurably better software quality indicators including lower defect rates and faster resolution.

III. SYSTEM OBJECTIVES AND ARCHITECTURE

A. Objectives

The primary objectives of the proposed system are:

- Design a fully automated CI/CD pipeline eliminating manual intervention in build, test, and deployment stages.
- Adopt Terraform IaC for automated, consistent, and repeatable cloud resource provisioning on Microsoft Azure.
- Containerize workloads with Docker and orchestrate with Kubernetes for portability and scalability.
- Configure Azure Monitor for real-time observability and proactive incident detection.
- Eliminate environment inconsistencies by provisioning all environments identically from one Terraform configuration.
- Demonstrate a scalable cloud-native architecture for enterprise-grade deployments.

B. Overall System Architecture

The proposed system follows a five-pillar architecture: Source Code Management, Continuous Integration, IaC Provisioning, Continuous Deployment, and Monitoring. These pillars are realized using GitHub, Azure DevOps Pipelines, Terraform, Microsoft Azure, Docker, Kubernetes, and Azure Monitor.

The architectural flow begins at the developer workstation, where application code and Terraform infrastructure configuration are committed to a single GitHub repository (GitOps model). Code push triggers the CI/CD pipeline: build and test → Terraform provision → application deploy → health monitoring → feedback to developer.

TABLE I. SYSTEM ARCHITECTURE LAYERS

Layer	Responsibility
Developer Layer	Code authoring, commit, push
Source Control	Version control, pipeline trigger
CI/CD Layer	Build, test, validate, orchestrate
IaC Provisioning	Terraform cloud resource creation
Cloud Platform	Azure hosting, networking, compute
Container Layer	Docker image build, K8s orchestration
Monitoring	Observability, alerting, feedback
Security	RBAC, Key Vault, NSG

IV. SYSTEM SPECIFICATION

A. Software Requirements

Table II summarizes the primary software components. The technology stack balances capability, cloud-native integration, and industry adoption.

TABLE II. SOFTWARE COMPONENTS SUMMARY

Tool/Technology	Purpose
GitHub / Git	Source control, pipeline trigger
Azure DevOps Pipelines	CI/CD automation
Terraform v1.5+	IaC provisioning on Azure
Microsoft Azure	Cloud hosting, managed services
Docker Engine 24.0+	Application containerization
Kubernetes (AKS) v1.27+	Container orchestration, scaling
Azure Monitor	Metrics, alerting, dashboards
Azure Key Vault	Secrets and certificate management
HashiCorp HCL 2.0	Terraform configuration language

B. Hardware Requirements

Developer workstations require a minimum Intel Core i5 processor, 8 GB RAM, and 256 GB SSD. Azure cloud infrastructure provisioned by Terraform includes a Standard_B2s VM (2 vCPU, 4 GB RAM) running Ubuntu Server 22.04 LTS, Standard Load Balancer, Virtual Network (10.0.0.0/16), Azure Container Registry (Basic SKU), AKS (Standard B2ms nodes), and Azure Key Vault. All resources are provisioned automatically through Terraform.

V. SYSTEM DESIGN

A. Data Flow

At context level, three external entities interact with the system: the Developer (code and configuration), Cloud Infrastructure — Azure (receives provisioning commands), and End Users (access the deployed application). At Level 1, data flows from developer code push through GitHub, triggering the Azure DevOps pipeline, which executes build and test stages before invoking Terraform. Infrastructure outputs (VM public IP, resource group) are passed as pipeline variables to the deployment stage. Azure Monitor aggregates health metrics and logs, returning feedback to the development team.

B. Pipeline State Diagram

The pipeline lifecycle: IDLE → TRIGGERED (code push) → BUILDING → TESTING → VALIDATING → PROVISIONING (Terraform) → DEPLOYING → MONITORING (health checks) → COMPLETED. On failure, the pipeline transitions to ROLLING_BACK (reverting to previous stable version) or FAILED. Successful completion returns to IDLE awaiting the next commit.

C. Existing System Analysis

The existing manual deployment model involves discrete handoffs between development, QA, and operations teams — each operating in isolation. Administrators manually

provision servers via cloud consoles, configure middleware through undocumented terminal commands, and copy application artifacts manually. Table III compares the existing system against the proposed solution.

TABLE III. EXISTING SYSTEM VS PROPOSED SOLUTION

Aspect	Existing	Proposed
Provisioning	Manual console (hrs)	Terraform (8–12 min)
Consistency	~70% (drift)	100% identical
Deployment Freq.	Weekly	Multiple/day
Failure Rate	~15%	<3%
Rollback	Manual, error-prone	Git + TF state
Monitoring	Separate, reactive	Integrated, proactive

VI. SYSTEM IMPLEMENTATION

A. Source Code and Repository Module

GitHub hosts both application source code and Terraform infrastructure configuration — a unified GitOps model ensuring infrastructure changes undergo the same review workflows as application code. The repository is organized as: root (Terraform files: main.tf, variables.tf, outputs.tf), /app (application source), /k8s (Kubernetes manifests), /.azure-pipelines (CI/CD YAML). Branch protection rules enforce peer review and CI checks before merging. Pull request workflows post terraform plan output as PR comments, enabling reviewers to inspect infrastructure changes before approval.

B. Continuous Integration Module

The CI Automation Module is defined in azure-pipelines.yml committed to the repository, structured into five sequential stages: (1) Build — installs Node.js dependencies via npm ci and generates production assets; (2) Test — executes Jest unit tests with coverage reporting in JUnit XML; (3) Code Quality — runs ESLint for JavaScript linting; (4) Infrastructure Validation — runs terraform validate and terraform fmt -check; (5) Security Scan — executes tfsec for Terraform security analysis. Azure Key Vault-linked variable groups supply pipeline secrets without hardcoded credentials.

C. Infrastructure as Code Provisioning Module

Terraform provisions all Azure resources declaratively: Resource Group, Virtual Network, Subnet, Network Security Group (HTTP/HTTPS/SSH rules), Public IP, Network Interface, Load Balancer, Backend Pool, Health Probe, and Linux Virtual Machine. Remote state is stored in Azure Blob Storage with versioning and Azure AD RBAC access control. Pipeline sequence: terraform init → terraform plan -out=tfplan → terraform apply tfplan. Saved plan files eliminate plan-to-apply race conditions. Terraform output commands extract provisioned resource attributes and pass them as pipeline variables to deployment stages.

D. Container and Deployment Module

In VM-based mode, the pipeline SSHs to the Azure VM, deploys artifacts, configures NGINX virtual host, and performs HTTP health checks. In Kubernetes mode, a multi-stage Dockerfile packages the application: Stage 1 (Node.js) installs dependencies and builds assets; Stage 2 (nginx:alpine) serves only built static files, minimizing image size. Images are tagged with Git commit SHA (immutable, traceable) and pushed to ACR. Kubernetes Deployment manifests use RollingUpdate with maxSurge: 1 and maxUnavailable: 0 for zero-downtime updates.

E. Monitoring and Security Modules

Azure Monitor collects platform metrics (CPU, memory, disk I/O, network) automatically from all provisioned resources. Log Analytics aggregates VM logs, Kubernetes pod logs, and pipeline execution logs. Alert rules fire on: pipeline failure, VM CPU >80% for 5 min, HTTP error rate >5%, and pod crash loops. Azure Key Vault centralizes credentials and API keys. NSG rules permit only ports 80, 443, and 22 from restricted source ranges. The pipeline service principal is scoped to Contributor on the specific Resource Group only, following the principle of least privilege.

VII. RESULTS AND DISCUSSION

The DevOps Automation Pipeline was validated through unit, integration, infrastructure, and acceptance test cycles comprising 30 defined test cases, all returning PASS results.

Pipeline Execution: End-to-end pipeline from code commit to production deployment completed in 12–18 minutes, compared to 3–5 days manually — a 95% lead time reduction. Terraform provisioned all Azure resources within 8–12 minutes. Idempotent apply confirmation showed zero resource modifications on second apply with unchanged configuration.

Kubernetes Deployment: Rolling updates replaced pod instances without service interruption, validated by continuous HTTP health checks. HorizontalPodAutoscaler scaled pod replicas from 2 to 8 within 90 seconds under simulated CPU load.

TABLE IV. PERFORMANCE BENCHMARK COMPARISON

Metric	Manual	Automated
Lead Time	3–5 days	12–18 min
Deploy Frequency	Weekly	Multi/day
Change Failure Rate	~15%	<3%
MTTR	4–8 hours	15–30 min
Infra Provision Time	2–4 hours	8–12 min
Env. Consistency	~70%	100%
Operational Cost	~20 hrs/release	<1 hr/release

VIII. APPLICATIONS AND BENEFITS

A. Key Applications

• Web Application Deployment: Node.js, Python, Java Spring Boot, .NET Core on Azure VM or App Service.

• Microservices: Independent deployment pipelines enabling team autonomy.

• Enterprise Delivery: Standardized pipelines across multiple teams and projects.

• FinTech: High-reliability deployment with automated rollback on failure.

• Healthcare: Compliant, auditable pipelines satisfying regulatory traceability.

• Multi-Cloud: Terraform's provider-agnostic model extends to AWS and GCP.

B. System Benefits

• Continuous Delivery: Automation eliminates deployment gaps from manual failures.

• Cost Optimization: Automated resource right-sizing reduces cloud spending.

• Scalability: Kubernetes HPA handles traffic spikes automatically.

• Audit Trail: All infrastructure and code changes tracked in Git history.

• Security by Design: RBAC, NSG, and Key Vault integrated following DevSecOps principles.

IX. CONCLUSION AND FUTURE WORK

This paper presented a DevOps Automation Pipeline integrated with Infrastructure as Code that automates the complete software delivery lifecycle from code commit to production deployment on Microsoft Azure. The system integrates GitHub source control, Azure DevOps CI/CD automation, Terraform IaC provisioning, Docker and Kubernetes containerization, and Azure Monitor observability into a cohesive, production-quality pipeline.

Experimental validation confirmed 95% reduction in deployment lead time, 80% reduction in change failure rate, and 90% reduction in mean time to recovery versus manual baselines. Terraform IaC ensures infrastructure is reproducible, auditable, and recoverable — the entire cloud environment can be reprovisioned from configuration in minutes.

Future enhancements include: (1) multi-region deployment with Azure Traffic Manager for geo-redundancy; (2) full GitOps using ArgoCD or FluxCD for pull-based Kubernetes reconciliation; (3) canary deployments with automated promotion/rollback via Flagger; (4) DevSecOps integration with SAST (SonarQube), DAST (OWASP ZAP), and dependency scanning (Snyk); (5) multi-cloud Terraform extension to AWS and GCP; and (6) AI-powered anomaly detection using Azure Sentinel.

X. REFERENCES

[1] IEEE Xplore, 'DevOps Automation Pipeline Deployment Using Infrastructure as Code,' Doc ID: 10910699, IEEE Xplore, 2024.

[2] J. Humble and D. Farley, Continuous Delivery. Addison-Wesley, 2010.

[3] M. Leppanen et al., 'The highways and country roads to continuous deployment,' IEEE Software, vol. 32, no. 2, pp. 64–72, 2015.

[4] DORA Research Program, 'Accelerate: State of DevOps Report,' Google Cloud, 2023.

[5] Y. Zhao et al., 'The impact of continuous integration on other software development practices,' Proc. ASE, 2017, pp. 60–71.

[6] M. Shahin et al., 'Continuous integration, delivery and deployment: A systematic review,' IEEE Access, vol. 5, pp. 3909–3943, 2017.

[7] K. Morris, Infrastructure as Code: Managing Servers in the Cloud. O'Reilly Media, 2016.

[8] A. Rahman et al., 'The seven sins: Security smells in infrastructure as code scripts,' Proc. ICSE, 2019, pp. 164–175.

[9] B. Burns et al., 'Borg, Omega, and Kubernetes,' ACM Queue, vol. 14, no. 1, pp. 70–93, 2016.

[10] D. Merkel, 'Docker: Lightweight Linux containers for consistent development and deployment,' Linux Journal, 2014.

[11] M. Hilton et al., 'Usage, costs, and benefits of continuous integration in open-source projects,' Proc. ASE, 2016, pp. 426–437.

[12] HashiCorp, 'Terraform Documentation: AzureRM Provider,' HashiCorp Developer, 2024.

[13] Microsoft Corporation, 'Azure DevOps Pipelines Documentation,' Microsoft Learn, 2024.

[14] The Kubernetes Authors, 'Kubernetes Documentation: Concepts,' Kubernetes.io, 2024.

[15] T. Kim et al., The DevOps Handbook. IT Revolution Press, 2016.

This work was carried out at Jayalakshmi Institute of Technology, Thoppur, Dharmapuri, Tamil Nadu, India under the supervision of Prof. A. Kannammal, M.E, Ph.D., Department of Information Technology. Manuscript received: April 2026 | Revised: April 2026 | Accepted: April 2026