Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Implementing Infrastructure as Code (IaC) with Terraform for Scalable Cloud Deployments

Naveen Kumar Kasarla
Independent Researcher, USA

ARTICLE INFO

Received: 12 Aug 2025

Revised: 19 Sept 2025

Accepted: 26 Sept 2025

ABSTRACT

Cloud infrastructure management through manual processes creates significant operational bottlenecks for engineering teams attempting to deliver software efficiently. Traditional provisioning requires submitting requests through ticketing systems and waiting extended periods while operations staff manually configure resources through cloud provider interfaces. Configuration inconsistencies emerge when different team members provision similar environments using varying procedures and timing. Manual setup practices produce configuration drift where development, staging, and production environments diverge from their intended specifications over time. Teams encounter deployment failures caused by subtle environmental differences that consume substantial debugging effort rather than productive feature development activities. Infrastructure as Code solves these operational problems by allowing teams to specify cloud resources through configuration files that describe what infrastructure should exist rather than how to build it. Terraform offers unified provisioning across different cloud platforms while integrating with version control systems that maintain historical records of infrastructure changes. This declarative framework removes dependencies on complex scripts that fail when cloud providers modify their programming interfaces. Version-controlled infrastructure definitions enable code review processes that identify configuration errors before deployment while providing audit trails for system modifications. Teams can implement rollback procedures for problematic changes using standard version control workflows. Organizations implementing Infrastructure as Code report improved deployment consistency and reduced provisioning times across multiple cloud environments while eliminating manual configuration errors that typically cause production incidents.

Keywords: Infrastructure As Code, Terraform Implementation, Cloud Automation, Scalable Deployments, DevOps Integration

1. Introduction

Setting up cloud infrastructure manually drives everyone crazy because teams waste months clicking through web interfaces instead of building products that customers actually want to buy. Engineers submit requests for new environments and wait three weeks while operations staff manually configure servers through tedious point-and-click procedures. Meanwhile, project deadlines slip and competitors ship features faster [1].

Every organization follows the same broken process. Developers need testing environments, so they create tickets and wait. Operations teams eventually get around to provisioning resources, but by then,

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

requirements have changed and everything needs rebuilding. Environments end up configured differently because different people handled setup at different times using slightly different procedures.

Microservices make this chaos exponentially worse. Applications now depend on dozens of interconnected services that need complex networking, service discovery, and load balancing configurations. Manually coordinating these dependencies across multiple cloud regions becomes impossible when dealing with hundreds of moving pieces that constantly change.

Configuration drift happens naturally when engineers make quick fixes directly in production without updating documentation. Development works fine, staging behaves strangely, and production breaks in mysterious ways because each environment diverged from its intended configuration over months of ad-hoc modifications.

Infrastructure as Code fixes these coordination problems by letting teams define what they want in simple configuration files instead of figuring out complicated provisioning steps. Terraform provides consistent interfaces across Amazon, Microsoft, and Google clouds while tracking changes through version control just like application code [3].

The shift from clicking interfaces to writing code eliminates the tribal knowledge problem, where only certain people know how to provision specific resources. Teams can review infrastructure changes before deployment while maintaining complete histories of what changed when things go wrong. Self-service environment creation becomes possible without waiting for operations tickets or manual coordination between different teams.

Challenge	Impact
Manual Configuration	Extended provisioning times and human error introduction
Configuration Drift	Environmental inconsistencies are causing deployment failures
Ticket-Based Workflows	Development bottlenecks and delayed feature delivery
Lack of Version Control	Inability to track changes or implement rollbacks
Provider-Specific Scripts	Vendor lock-in and maintenance overhead
Knowledge Silos	Operational dependencies on individual expertise

Table 1: Infrastructure Provisioning Challenges [1,3]

1.1 Terraform Architecture Components

Terraform breaks infrastructure management into separate pieces that handle different parts of cloud deployment without creating a monolithic system that becomes impossible to debug. Configuration files use HashiCorp's domain-specific language that reads almost like English rather than requiring complex programming knowledge. Teams describe what infrastructure they want instead of writing scripts that specify step-by-step provisioning procedures [2].

State files track what actually exists in cloud accounts versus what configuration files say should exist. This comparison enables Terraform to figure out what changes are needed without accidentally destroying existing resources or creating duplicates. Remote state storage prevents the classic problem where multiple engineers step on each other's changes when working on shared infrastructure simultaneously.

Provider plugins translate Terraform's generic syntax into cloud-specific API calls that actually create resources. Identical configuration syntax works across Amazon, Microsoft, and Google platforms because provider plugins translate generic Terraform code into platform-specific API calls automatically. This flexibility prevents vendor dependency while enabling cost-effective workload distribution across multiple cloud providers for better pricing and redundancy planning.

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Resource blocks define individual infrastructure components like servers, databases, and networking equipment using consistent syntax regardless of which cloud provider hosts them. Dependencies get resolved automatically through Terraform's internal graph system that figures out creation order without requiring manual sequencing [7].

Variables enable reusable configurations that adapt to different environments without copying and pasting code everywhere. Teams create modules that accept parameters for customizing infrastructure characteristics while maintaining architectural consistency across multiple deployments and environments.

Output values expose information about created resources for consumption by other Terraform configurations or external systems. This modularity enables different teams to manage separate infrastructure components that integrate through well-defined interfaces rather than requiring centralized management of everything.

Component	Function
Configuration Files	Define desired infrastructure state declaratively
State Management	Track resource relationships and current deployment status
Provider Plugins	Interface with cloud platforms and services
Resource Blocks	Specify individual infrastructure components
Variable Definitions	Enable parameterized and reusable configurations
Output Values	Expose resource information for external consumption

Table 2: Terraform Core Components [2,7]

1.2 Multi-Cloud Provisioning Strategies

Microservices architectures create networking nightmares that nobody talks about until deployment day arrives. Suddenly, teams discover their containerized applications need service meshes, load balancers, and discovery mechanisms spread across different cloud providers. Manual coordination breaks down completely when dealing with hundreds of services that change constantly based on business requirements and scaling demands [4]. Each cloud provider handles networking differently, so teams end up learning Amazon's VPC quirks, Azure's virtual network limitations, and Google Cloud's firewall peculiarities. Engineers waste months figuring out how to make identical applications work across different platforms while maintaining security policies that satisfy compliance auditors.

Strategy	Application
Provider Abstraction	Unified configuration syntax across different clouds
Resource Standardization	Consistent naming and tagging conventions
Environment Isolation	Separate state files for different deployment stages
Geographic Distribution	Regional deployments for latency optimization
Disaster Recovery	Cross-cloud backup and failover capabilities
Cost Optimization	Dynamic resource allocation based on pricing

Table 3: Multi-Cloud Deployment Strategies [4,8]

Geographic distribution becomes essential when customers complain about slow response times from distant data centers. Teams need identical infrastructure deployed across multiple continents without spending years configuring each region manually. Terraform modules enable this replication while

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

accounting for regional differences in available instance types and regulatory requirements. Environment separation prevents the classic problem where someone accidentally destroys production while testing configuration changes in development. Different state files and workspace isolation ensure that experimental modifications stay contained within appropriate boundaries. Teams can experiment freely without risking business-critical infrastructure during normal development cycles [8]. Cost management gets interesting when teams can shift workloads between providers based on current pricing and resource availability. Some applications run cheaper on AWS while others benefit from Azure's specialized services. Terraform configurations enable dynamic provider selection based on cost models and performance requirements rather than vendor lock-in decisions.

Disaster recovery planning becomes realistic when infrastructure exists across multiple cloud providers automatically. Regional outages happen regularly, so having standby infrastructure ready for immediate activation prevents those embarrassing downtime incidents that make headlines. Terraform automation coordinates failover procedures faster than manual emergency response protocols.

2. State Management and Collaboration

State files create the biggest headache in team Terraform deployments because everyone needs access to the current infrastructure status without stepping on each other's changes. Local state files work fine for individual experimentation, but become disasters when multiple engineers try to modify shared infrastructure simultaneously. Someone inevitably overwrites critical state information or creates resource conflicts that require manual intervention [1].

Remote state backends solve collaboration problems by storing state files in shared locations like Amazon S3 buckets or Terraform Cloud workspaces. Teams can configure automatic state locking that prevents concurrent modifications while ensuring everyone works with the current infrastructure status. This centralized approach eliminates the classic problem where engineers accidentally destroy resources because they were working with outdated state information.

State file corruption represents a serious operational risk because losing the state means losing track of existing infrastructure entirely. Remote backends provide versioning and backup capabilities that enable recovery when state files get corrupted or accidentally deleted. Teams can roll back to previous state versions when deployments go wrong or infrastructure modifications create unexpected problems.

Workspace isolation enables different teams to manage separate environments through distinct state boundaries while sharing common Terraform configurations. Development, staging, and production environments maintain independent state files that prevent accidental cross-environment modifications. This separation allows autonomous team management while preserving architectural consistency through shared modules and standardized deployment procedures.

State inspection capabilities help troubleshoot deployment problems by providing detailed visibility into resource relationships and metadata. Teams can examine the current state without making modifications while understanding how Terraform tracks dependencies between different infrastructure components during planning and application phases.

2.1 CI/CD Pipeline Integration

Automated pipeline integration transforms Terraform deployments from manual procedures into systematic workflows that include testing, validation, and approval gates before infrastructure modifications reach production environments. Jenkins, GitLab, and GitHub Actions provide pipeline frameworks that execute Terraform commands automatically while capturing detailed logs of all deployment activities [5].

Pipeline stages typically include initialization, validation, planning, and application phases that mirror manual Terraform workflows while adding automated testing and approval controls. Teams can configure

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Terraform plan outputs as pipeline artifacts that show proposed changes before human reviewers approve actual infrastructure modifications.

Pipeline previews eliminate unexpected infrastructure changes by showing exactly what modifications will occur before human reviewers approve deployment execution. Configuration validation through TFLint and Terratest identifies syntax problems, policy violations, and architectural inconsistencies during build phases rather than after resources reach production environments. These automated checks enforce naming conventions, tagging requirements, and security standards consistently across all deployments without manual review overhead.

Secret management becomes critical when pipelines need access to cloud provider credentials and sensitive configuration values. HashiCorp Vault, AWS Secrets Manager, and similar tools provide secure credential storage that prevents hardcoding sensitive information in Terraform configurations or pipeline definitions [6].

Deployment approvals enable human oversight for critical infrastructure changes while allowing automatic deployment of routine modifications like scaling adjustments or configuration updates. Teams can configure approval requirements based on change scope, environment criticality, and business impact while maintaining deployment velocity for non-critical modifications.

Pattern	Implementation
Pipeline Automation	Automated Terraform plan and apply stages
Change Approval	Human review gates for production deployments
Testing Integration	Validation checks before infrastructure changes
Rollback Mechanisms	Automated reversion for failed deployments
Environment Promotion	Progressive deployment across staging environments
Secret Management	Secure handling of credentials and sensitive data

Table 4: CI/CD Integration Patterns [5,6]

2.2 Security and Compliance Frameworks

Automated infrastructure deployment creates security blind spots where traditional approval workflows used to catch policy violations before resources reach production environments. Teams need programmatic validation through policy-as-code tools that enforce organizational security standards without slowing down deployment velocity [4].

Permission management becomes complex when balancing developer self-service needs against security boundaries that prevent unauthorized access to critical infrastructure components. Identity systems must enable appropriate autonomy while maintaining oversight capabilities that satisfy security requirements across different operational environments. Industry compliance demands continuous monitoring because infrastructure modifications can introduce regulatory violations that remain undetected until formal audit periods. Automated configuration analysis identifies non-compliant resources while generating documentation that regulatory frameworks require for ongoing compliance verification.

Data protection requires systematic encryption enforcement across all infrastructure components because inconsistent manual implementation creates vulnerability gaps. Standardized security policies can be embedded automatically into Terraform configurations without requiring specialized security knowledge from development teams. Forensic capabilities depend on comprehensive change tracking that records infrastructure modification details for security incident response and regulatory investigations. Audit

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

systems must maintain searchable historical records that identify specific actors, timing, and affected resources during compliance reviews or security breach analysis [8].

3. Operational Efficiency Metrics

Organizations implementing intelligent operations report substantial improvements in operational metrics that directly impact business performance and customer satisfaction. Mean time to detection decreases dramatically when automated pattern recognition identifies developing problems before they escalate into service disruptions. Teams catch issues during early development phases rather than after customers experience degraded performance or complete service outages [1]. Resolution times improve significantly as automated correlation eliminates hours of manual investigation that previously consumed engineering resources during critical incidents. Engineers spend less time hunting through disparate monitoring tools and log files while automated systems provide targeted insights about failure origins and recommended remediation strategies. This efficiency gain allows teams to restore services faster while reducing the business impact associated with extended outages. False positive reduction transforms operational workflows by eliminating alert fatigue that overwhelms monitoring teams with irrelevant notifications. Traditional threshold-based monitoring generates thousands of alerts daily, most representing normal system variations rather than genuine problems requiring immediate attention. Intelligent filtering reduces alert volumes by identifying patterns that distinguish routine operational changes from actual threats requiring investigation. Resource optimization develops naturally when teams transition from emergency response patterns toward strategic capacity management and architectural improvement activities. Engineering staff who previously handled constant operational crises can redirect attention toward system enhancements, performance optimization, and scaling strategies that address root causes of recurring problems. This forward-thinking orientation creates cumulative improvements where systems gain resilience and operational efficiency through sustained development efforts [12]. Operational cost reduction occurs through multiple mechanisms, including reduced overtime expenses, decreased infrastructure waste, and improved resource allocation efficiency. Automated incident response eliminates night-shift escalations for routine problems while predictive analytics enable right-sized infrastructure provisioning that avoids both over-provisioning costs and under-provisioning performance problems.Team productivity increases as engineers develop expertise in strategic system design rather than spending careers managing operational crises. Knowledge retention improves when institutional wisdom gets captured in automated playbooks and correlation rules rather than remaining trapped in individual experience. This systematization enables more consistent incident response while reducing dependencies on specific team members during critical operational periods.

Metric	Improvement Area
Deployment Speed	Reduced provisioning time through automation
Configuration Consistency	Eliminated environmental drift across stages
Change Visibility	Enhanced audit trails through version control
Error Reduction	Decreased manual configuration mistakes
Team Productivity	Improved collaboration through shared definitions
Infrastructure Reliability	Increased system stability through standardization

Table 5: Operational Efficiency Metrics [1,8]

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Conclusion

Automated provisioning cuts out the manual setup tasks that waste engineering time and slow down product delivery. Teams stop burning weeks on repetitive infrastructure work and start building features that customers actually care about using. Team dynamics change fundamentally when infrastructure stops being mysterious operations, knowledge that only certain people understand. Everyone can read configuration files and propose changes through normal code review processes. Version control shows exactly what happened when deployments break, making troubleshooting faster and less stressful. Success requires mental shifts from clicking cloud provider interfaces to writing declarative specifications. Teams must build workflows where automation handles predictable provisioning tasks while humans make judgment calls on security policies and compliance requirements that need business context. Terraform keeps evolving with new cloud providers and deployment features, but core concepts stay stable. Infrastructure becomes code that deploys consistently across different environments without requiring manual configuration each time. Teams that master this gain speed advantages over competitors who are still doing everything manually.

References

[1] Venkat Marella, "Implementing Infrastructure as Code (IaC) for Scalable DevOps Automation in Hybrid Cloud," Journal of Sustainable Solutions, ResearchGate, Dec. 2024. https://www.researchgate.net/publication/387058455_Implementing_Infrastructure_as_Code_IaC_for_Scalable_DevOps_Automation_in_Hybrid_Cloud
[2] Naga Murali Krishna Koneru, "Infrastructure as Code (IaC) for Enterprise Applications: A Comparative Study of Terraform and CloudFormation," American Journal of Technology, ResearchGate, May 2025.
https://www.researchgate.net/publication/391714883_Infrastructure_as_Code_IaC_for_Enterprise_Applications_A_Comparative_Study_of_Terraform_and_CloudFormation
[3] Sachin Sudhir Shinde, "Implementing infrastructure as code with Terraform for cloud-based services," World Journal of Advanced Engineering Technology and Sciences, Jun. 2025.
https://journalwjaets.com/sites/default/files/fulltext_pdf/WJAETS-2025-1161.pdf
[4] Perumalsamy Ravindran, "Automating Multi-Cloud Infrastructure: Leveraging Terraform and IaC for Scalable, Secure, and Efficient Cloud Management," IJSRCSEIT, Mar. 2025.
https://ijsrcseit.com/index.php/home/article/view/CSEIT25112704
[5] Taiwo Joseph Akinbolaji et al., "Automation in Cloud-Based DevOps: A Guide to CI/CD Pipelines and Infrastructure as Code (IaC) with Terraform and Jenkins," WJAETS, Nov. 2024.
https://wjaets.com/sites/default/files/WJAETS-2024-0542.pdf
[6] "Continuous Integration and Continuous Deployment Pipeline Automation Using AWS, Jenkins, Ansible, Terraform, Docker, Grafana, Prometheus," International Journal of Research Publication and Reviews, Feb. 2024.
https://ijrpr.com/uploads/V5ISSUE2/IJRPR22767.pdf
[6] "Continuous Integration and Continuous Deployment Pipeline Automation Using AWS, Jenkins, Ansible, Terraform, Docker, Grafana, Prometheus," International Journal of Core Engineering & Management, 2021.
https://ijcem.in/wp-content/uploads/2024/08/INFRASTRUCTURE-AS-CODE-IAC-BEST-PRACTICES-USING-TERRAFORM-OR-AWS-CLOUDFORMATION-FOR-MACHINE-LEARNING-1.pdf
[7] Zoe Vasileiou et al., "A knowledge-based approach for guided development of Infrastructure as Code," Springer Nature Link, Jun. 2025.

https://link.springer.com/article/10.1007/s10270-025-01294-1
[8] Rajkumar Kyadasu et al., "Exploring Infrastructure as Code Using Terraform in Multi-Cloud Deployments," Journal of Quantum Science and Technology (JQST), Dec. 2024.
https://www.jqst.org/index.php/j/article/view/94

Appendices

Appendix A: Terraform VPC Module

hcl

```
# modules/vpc/main.tf
resource "aws_vpc" "main" {
  cidr_block           = var.vpc_cidr
  enable_dns_hostnames = true
  enable_dns_support   = true

  tags = {
    Name        = var.vpc_name
    Environment = var.environment
  }
}

resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id

  tags = {
    Name = "${var.vpc_name}-igw"
  }
}

resource "aws_subnet" "public" {
  count             = length(var.public_subnets)
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.public_subnets[count.index]
  availability_zone = var.availability_zones[count.index]

  map_public_ip_on_launch = true

  tags = {
    Name = "${var.vpc_name}-public-${count.index + 1}"
    Type = "Public"
  }
}

# modules/vpc/variables.tf
variable "vpc_cidr" {
  description = "CIDR block for VPC"
  type        = string
```

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

```
 default     = "10.0.0.0/16"
}

variable "vpc_name" {
 description = "Name of the VPC"
 type        = string
}

variable "environment" {
 description = "Environment name"
 type        = string
}

variable "public_subnets" {
 description = "Public subnet CIDR blocks"
 type        = list(string)
}

variable "availability_zones" {
 description = "Availability zones"
 type        = list(string)
}
```

☐**Appendix B: EC2 Instance Module**

hcl

```
☐# modules/ec2/main.tf
resource "aws_instance" "instance" {
 ami           = var.ami_id
 instance_type = var.instance_type
 subnet_id     = var.subnet_id

 vpc_security_group_ids = var.security_group_ids
 key_name               = var.key_name

 user_data = var.user_data

 root_block_device {
  volume_type = var.root_volume_type
  volume_size = var.root_volume_size
  encrypted   = true
 }

 tags = merge(
  var.tags,
  {
   Name = var.instance_name
  }
```

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

```
 )
}

resource "aws_eip" "instance_eip" {
 count    = var.associate_public_ip ? 1 : 0
 instance = aws_instance.instance.id
 domain   = "vpc"

 tags = {
  Name = "${var.instance_name}-eip"
 }
}

# modules/ec2/variables.tf
variable "ami_id" {
 description = "AMI ID for the instance"
 type        = string
}

variable "instance_type" {
 description = "Instance type"
 type        = string
 default     = "t3.micro"
}

variable "subnet_id" {
 description = "Subnet ID where instance will be launched"
 type        = string
}

variable "security_group_ids" {
 description = "List of security group IDs"
 type        = list(string)
}

variable "key_name" {
 description = "Key pair name"
 type        = string
}

variable "instance_name" {
 description = "Name tag for the instance"
 type        = string
}

variable "associate_public_ip" {
```

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

```
  description = "Associate public IP with instance"
  type        = bool
  default     = false
}

variable "user_data" {
  description = "User data script"
  type        = string
  default     = ""
}

variable "root_volume_type" {
  description = "Root volume type"
  type        = string
  default     = "gp3"
}

variable "root_volume_size" {
  description = "Root volume size in GB"
  type        = number
  default     = 20
}

variable "tags" {
  description = "Tags to apply to resources"
  type        = map(string)
  default     = {}
}
```

☐Appendix C: Jenkins Pipeline Configuration

```
groovy
☐pipeline {
    agent any

    parameters {
        choice(
            name: 'ACTION',
            choices: ['plan', 'apply', 'destroy'],
            description: 'Select Terraform action'
        )
        string(
            name: 'WORKSPACE',
            defaultValue: 'dev',
            description: 'Terraform workspace'
        )
    }
```

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

```
  environment {
    TF_VAR_environment = "${params.WORKSPACE}"
    AWS_DEFAULT_REGION = 'us-west-2'
  }

  stages {
    stage('Checkout') {
      steps {
        checkout scm
      }
    }

    stage('Terraform Init') {
      steps {
        sh '''
          terraform init \
            -backend-config="bucket=terraform-state-bucket" \
            -backend-config="key=${WORKSPACE}/terraform.tfstate" \
            -backend-config="region=${AWS_DEFAULT_REGION}"
        '''
      }
    }

    stage('Terraform Workspace') {
      steps {
        sh '''
          terraform workspace select ${WORKSPACE} || terraform workspace new ${WORKSPACE}
        '''
      }
    }

    stage('Terraform Plan') {
      steps {
        sh '''
          terraform plan -out=tfplan-${BUILD_NUMBER} -var-
file="environments/${WORKSPACE}.tfvars"
        '''
        archiveArtifacts artifacts: 'tfplan-*', fingerprint: true
      }
    }

    stage('Terraform Apply') {
      when {
        expression { params.ACTION == 'apply' }
      }
      steps {
```

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

```
            input message: 'Apply Terraform changes?', ok: 'Apply'
            sh '''
              terraform apply tfplan-${BUILD_NUMBER}
            '''
          }
        }

        stage('Terraform Destroy') {
          when {
            expression { params.ACTION == 'destroy' }
          }
          steps {
            input message: 'Destroy infrastructure?', ok: 'Destroy'
            sh '''
              terraform destroy -auto-approve -var-file="environments/${WORKSPACE}.tfvars"
            '''
          }
        }
      }

      post {
        always {
          cleanWs()
        }
        failure {
          emailext (
            subject: "Terraform Pipeline Failed: ${env.JOB_NAME} - ${env.BUILD_NUMBER}",
            body: "The Terraform pipeline has failed. Please check the build logs.",
            to: "${env.CHANGE_AUTHOR_EMAIL}"
          )
        }
      }
    }
```

☐Appendix D: Terraform State Management Configuration

```
hcl
☐# backend.tf
terraform {
  required_version = ">= 1.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
```

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

```
  backend "s3" {
    bucket         = "terraform-state-bucket"
    key            = "infrastructure/terraform.tfstate"
    region         = "us-west-2"
    encrypt        = true
    dynamodb_table = "terraform-state-lock"
  }
}

# main.tf
provider "aws" {
  region = var.aws_region

  default_tags {
    tags = {
      Project     = var.project_name
      Environment = var.environment
      ManagedBy   = "Terraform"
    }
  }
}

# State bucket and locking table
resource "aws_s3_bucket" "terraform_state" {
  bucket        = "terraform-state-bucket"
  force_destroy = false

  lifecycle {
    prevent_destroy = true
  }
}

resource "aws_s3_bucket_versioning" "terraform_state" {
  bucket = aws_s3_bucket.terraform_state.id
  versioning_configuration {
    status = "Enabled"
  }
}

resource "aws_s3_bucket_server_side_encryption_configuration" "terraform_state" {
  bucket = aws_s3_bucket.terraform_state.id

  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
```

Journal of Information Systems Engineering and Management
2025, 10(60s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

```
  }
}

resource "aws_dynamodb_table" "terraform_state_lock" {
  name         = "terraform-state-lock"
  billing_mode   = "PAY_PER_REQUEST"
  hash_key      = "LockID"

  attribute {
    name = "LockID"
    type = "S"
  }
}

# variables.tf
variable "aws_region" {
  description = "AWS region"
  type       = string
  default     = "us-west-2"
}

variable "project_name" {
  description = "Name of the project"
  type       = string
}

variable "environment" {
  description = "Environment name"
  type       = string
}
```

□